A MARSHALL CAVENDISH **16** COMPUTER COURSE IN WEEKLY PARTS

INPUT

LEARN PROGRAMMING - FOR FUN AND THE FUTURE



UK £1.00
Republic of Ireland £1.25
Malta 85c
Australia $2.25
New Zealand $2.95

INPUT

Vol. 2 **No 16**

INDEX

The last part of INPUT, Part 52, will contain a complete, cross-referenced index. For easy access to your growing collection, a cumulative index to the contents of each issue is contained on the inside back cover.

PICTURE CREDITS
Front cover, Dave King. Pages 477, 479, 480, 482, Barry Thorpe. Pages 484, 485, Jeremy Gower. Page 486, Kuo Kang Chen. Pages 485, 487, 488, Ray Duns. Page 490, Chris Lyon. Pages 492, 494, 496, Andrew MacConville. Pages 498, 500, 502, Dave King. Pages 504, 506, Lionel Jeans.

Published by Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA, England. Printed by Artisan Presss, Leicester and Howard Hunt Litho, London.



There are four binders each holding 13 issues.

HOW TO ORDER YOUR BINDERS

UK and Republic of Ireland:
Send £4.95 (inc p & p) (IR£5.95) for each binder to the address below:
Marshall Cavendish Services Ltd,
Department 980, Newtown Road,
Hove, Sussex BN3 7DN

Australia: See inserts for details, or write to INPUT, Times Consultants, PO Box 213, Alexandria, NSW 2015

New Zealand: See inserts for details, or write to INPUT, Gordon and Gotch (NZ) Ltd, PO Box 1595, Wellington

Malta: Binders are available from local newsagents.

BACK NUMBERS

Back numbers are supplied at the regular cover price (subject to availability).

UK and Republic of Ireland:
INPUT, Dept AN, Marshall Cavendish Services,
Newtown Road, Hove BN3 7DN

Australia, New Zealand and Malta:
Back numbers are available through your local newsagent.

COPIES BY POST

Our Subscription Department can supply copies to any UK address regularly at £1.00 each. For example the cost of 26 issues is £26.00; for any other quantity simply multiply the number of issues required by £1.00. Send your order, with payment to:

Subscription Department, Marshall Cavendish Services Ltd,
Newtown Road, Hove, Sussex BN3 7DN

Please state the title of the publication and the part from which you wish to start.

HOW TO PAY: Readers in UK and Republic of Ireland: All cheques or postal orders for binders, back numbers and copies by post should be made payable to:
Marshall Cavendish Partworks Ltd.

QUERIES: When writing in, please give the make and model of your computer, as well as the Part No., page and line where the program is rejected or where it does not work. We can only answer specific queries – and please do not telephone. Send your queries to INPUT Queries, Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA.

INPUT IS SPECIALLY DESIGNED FOR:

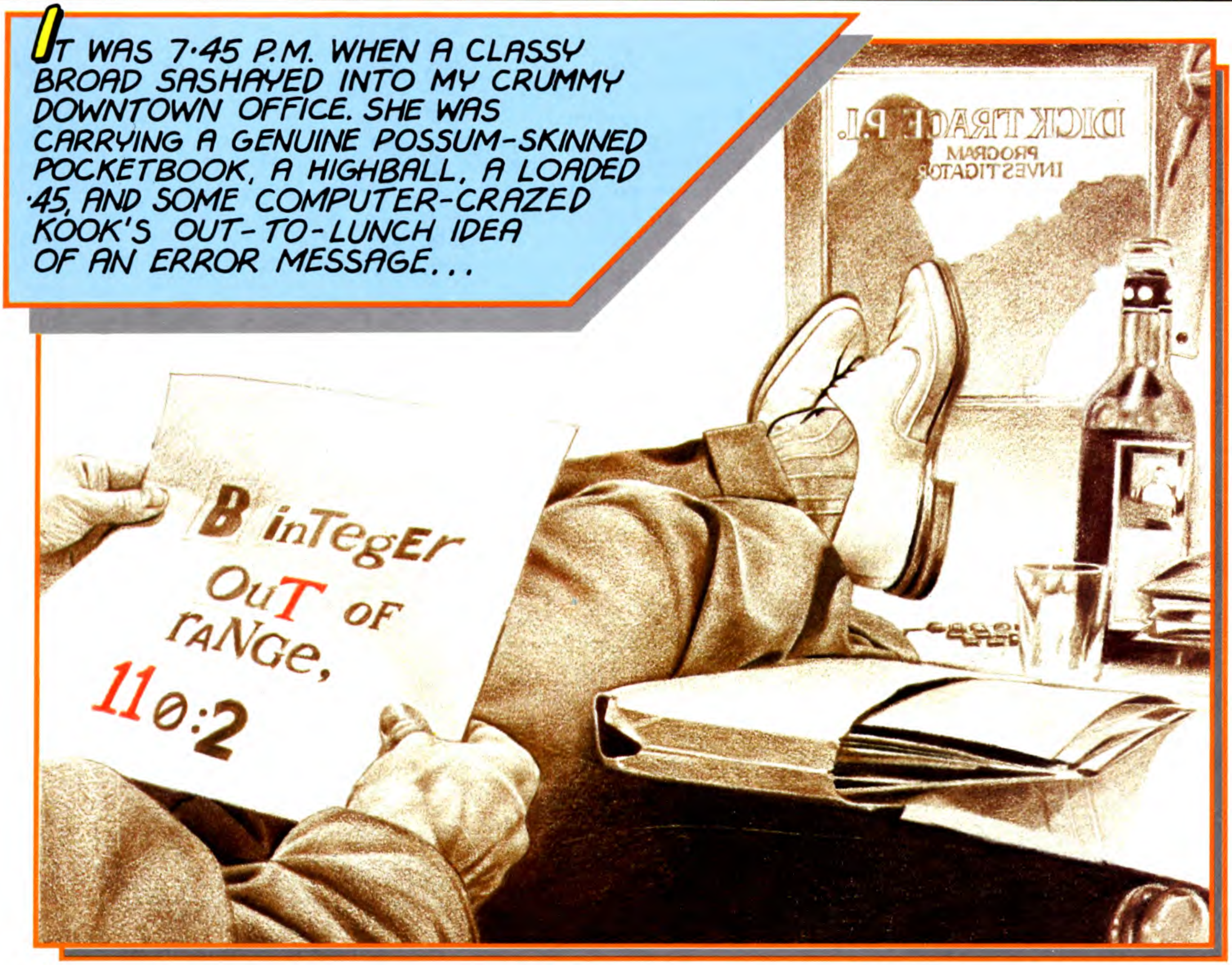
The SINCLAIR ZX SPECTRUM (16K, 48K, 128 and +), COMMODORE 64 and 128, ACORN ELECTRON, BBC B and B+, and the DRAGON 32 and 64.

In addition, many of the programs and explanations are also suitable for the SINCLAIR ZX81, COMMODORE VIC 20, and TANDY COLOUR COMPUTER in 32K with extended BASIC. Programs and text which are specifically for particular machines are indicated by the following symbols:

SPECTRUM 16K, 48K, 128, and +
COMMODORE 64 and 128
ACORN ELECTRON, BBC B and B+
DRAGON 32 and 64
ZX81
VIC 20
TANDY TRS80 COLOUR COMPUTER

SPECTRUM TRACE PROGRAM

- WHAT THE TRACE CAN DO
- FINDING THE ERRORS
- ENTERING THE PROGRAM
- HOW IT WORKS
- HOW TO USE IT



Track down those elusive bugs and evasive errors in your programs, and investigate what's going wrong with this powerful Spectrum trace program for 48K machines

It is almost impossible to key in a long program—like your assembler—without introducing some errors. No matter how much it is checked, there are some bugs that defy even the deftest programmer, without the aid of some powerful diagnostic tool.

Having a properly working assembler is essential. Many of the following chapters depend on it and it is vital that you locate all of the bugs in it now. So *INPUT* is providing Spectrum owners with a trace program to help them check their assemblers out. The Dragon and Tandy have trace programs built in. So do the BBC and Electron. A trace for the Commodore will be given next time.

The trace programs listed below are given in assembly language as well as machine code. If your assembler is not working you can feed in the trace program machine code using the machine code monitor given on pages 280 and 281. If your assembler is working, you can assemble the trace and SAVE it so that you can use it to diagnose problems in other BASIC programs that you have written. And if you are not sure whether your assembler is working or not, you can test it by trying to assemble the trace program.

HOW TO USE IT

When a BASIC program will not RUN, your computer will often give you an error message which tells you which line it cannot execute.

This may be all you need to know to debug a short, simple program. But when your programs get longer and more complicated, such a message may still leave you in the dark. A particular line may be executed a number of times while the program is being RUN. And other lines that have been executed before it may set the variables to values that cause problems in the line which your computer eventually falters on.

This trace program simply PRINT out on the screen the number of each line—and, on the Spectrum, the number of the statement in that line—as it is executed. To make full use of the trace you should have a copy of the program handy—either your own printout or the version published in *INPUT*. Then follow the program through to the point where it stops, using the trace. This way you will be able to see clearly the structure of the program. You'll be able to spot whether the computer is RETURNing from subroutines properly. You'll also be able to work out the value of the variables as you go and check that conditional IF ... THEN statements are being fulfilled and that GOTOs go to the right line.

HOW IT WORKS

A trace program is rather special. It runs while another program—the BASIC program that you are checking out—is RUNning. Normally it is not possible to run two programs in your computer at the same time. In this case though, the two programs—although they seem to be running simultaneously—are not. The trace uses what are called *interrupt driven routines*.

These interrupt the main program every 50th of a second on the Spectrum. While the main program is halted for a fraction of the second, the interrupt driven routine is performed. And when it is finished, the main program RUNs again until the next interrupt.

BASIC programs are always interrupted when they are RUN. The computer breaks off every 50th of a second to scan the keyboard and to check to see if a key has been pressed. Interrupt driven routines are simply tacked onto this keyboard scan routine. Note that this will not work properly with an Interface 1 connected.

A long line of BASIC may be interrupted several times during its execution. So the trace may give you a line number repeated several times. Conversely, if a line is very short—say a single PRINT or a RETURN from a subroutine which takes less than a 50th of a second—there is a slight chance that the trace will miss it. If the trace does not list the number of a short line, try adding a delay—a FOR ... NEXT loop or a REM statement—to it.

S

The Spectrum has special problems when it comes to PRINTing on the screen during an interrupt routine. It has discrete output channels for PRINTing on the bottom of the screen and the top. And changing channel while a program is RUNning can lead to problems.

The way round this is to POKE the line number directly onto the screen. As it takes eight bytes to PRINT just one character, to simplify the PRINT procedure the line numbers are all PRINTed in the same place, out of the area BASIC wants to use. Alongside it is PRINTed the number of the statement within the line that the program is executing.

The trace routine is not called from BASIC. It is called from another machine code routine, which itself is called from BASIC. And it is this call routine which redirects the interrupts to the start address of the trace program itself.

Don't forget to CLEAR down to at least 65109 before you enter the following program. If you can enter this program using your assembler the origins are listed here. And even if it does not assemble properly you can check the translation of each instruction and identify bugs that way. If your assembler will not work at all, use your machine code monitor to input the program. The start address is 65110.

```
org 65110
ld a,9                   3E 09
ld i,a                   ED 47
im 2                     ED 5E
ret                      C9
org 65120                00 00 00
ld a,62                  3E 3E
ld i,a                   ED 47
im 1                     ED 56
ret                      C9
org 65129                00 00
rst 56                   FF
push af                  F5
ld a,(23622)             3A 46 5C
bit 7,a                  CB 7F
jr z,go                  28 02
pop af                   F1
ret                      C9
go di                    F3
push bc                  C5
push de                  D5
push hl                  E5
push ix                  DD E5
ld de,20726              11 F6 50
ld (posn),de             ED 53 FE FE
ld hl,(23621)            2A 45 5C
call lineno              CD B5 FE
ld de,20731              11 FB 50
ld (posn),de             ED 53 FE FE
ld hl,(23623)            2A 47 5C
ld h,0                   26 00
call statno              CD BB FE
ld hl,23286              21 F6 5A
ld (hl),71               36 47
ld de,23287              11 F7 5A
ld bc,9                  01 09 00
ldir                     ED B0
keylp ld a,127           3E 7F
in a,254                 DB FE
or 224                   F6 E0
cp 252                   FE FC
jr z,keylp               28 F6
pop ix                   DD E1
pop hl                   E1
pop de                   D1
pop bc                   C1
pop af                   F1
ei                       FB
ret                      C9
lineno ld bc,-1000       01 18 FC
call prt                 CD CE FE
statno ld bc,-100        01 9C FF
call prt                 CD CE FE
ld bc,-10                01 F6 FF
call prt                 CD CE FE
ld bc,-1                 01 FF FF
call prt                 CD CE FE
ret                      C9
prt xor a                AF
prtlp add hl,bc          09
inc a                    3C
jr c,prtlp               38 FC
sbc hl,bc                ED 42
dec a                    3D
add a,48                 C6 30
push hl                  E5
call print               CD E8 FE
ld hl,posn               21 FE FE
inc (hl)                 34
ld hl,(posn)             2A FE FE
call prtout              CD F5 FE
pop hl                   E1
ret                      C9
print ld bc,(23606)      ED 4B 36 5C
ld h,0                   26 00
ld l,a                   6F
add hl,hl                29
add hl,hl                29
add hl,hl                29
add hl,bc                09
ex de,hl                 EB
ret                      C9
prtout ld b,8            06 08
loop ld a,(de)           1A
ld (hl),a                77
inc h                    24
inc de                   13
djnz loop                10 FA
ret                      C9
posn defw 0              00 00
```

Note that there are three programs here, not just one. Each one has its own separate origin. The first program, which starts at 65,110, turns the main program on. The second, which starts at 65,120, turns the main program off. And the third, which starts at 65,129, does the actual trace.

The instructions **ld a,9** and **ld i,a** load the I register with 9. There is no instruction to load the I register directly with a number. The 9 is taken as the high byte of the two-byte interrupt vector, while the low byte is supplied by the Spectrum itself and is normally 255. So the interrupt vector points to 9*256 + 255, which equals 2,559. The pointer contained in 2,559 and 2,560 redirects it to 65,129, which is the start of the main program.

This may seem rather round about, but interrupt routines must be addressed indirectly. And if you tried to load the I register with a number greater than 64, to direct it into RAM where you could POKE in your own pointer, the screen characters start breaking up.

The **im 2** then changes the interrupt mode so that you can interfere with it.

The routine that turns the main program off loads the I register with 62 to return it to what it was before the vector was altered. And the **im 1** returns the interrupt mode to normal.

The main program starts with the instruction **rst 56**. This makes the microprocessor perform its normal interrupt routine—scan the keyboard and update the system variable clock. If you don't ensure that the keyboard is scanned, it is disabled and you can't edit BASIC program lines.

Whenever you do an interrupt routine it is essential that the contents of all the registers are exactly the same after you have finished the routine as when you started it. The only way to ensure this is to push the contents of all the registers onto the stack at the beginning of the program, and pop them all off again at the



end. The contents of BC, DE, HL and IX are all pushed onto the stack. But the contents of AF are pushed on first because the accumulator and the flag register are used to test whether a program is being RUN. If no BASIC program is being RUN there is no point in doing a trace.

The test is executed by loading the accumulator with the contents of memory location 23,622. Locations 23,621 and 23,622 contain the number of the current line of BASIC being executed. Although BASIC line numbers are stored in the high-low format in the BASIC program area, here they are stored low-high, as usual.

If no program is RUNning, though, these two locations fill up with a number far too high for a line number—9,999 is the highest number the Spectrum will take. So the contents of the high byte, in 23,622, are loaded into the accumulator and the instruction **bit 7,a** looks at the most significant bit. If this is 1, no BASIC program is RUNning. The **bit** instruction looks at the bits of a particular memory location or register. For example **bit 4,a** tests the fourth bit of the A register.

If the most significant bit of the most significant bytes of the current BASIC line number system variable is 0, the zero flag is set and **jr z,go** jumps to the next occurrence of the label **go**, which is at the start of the main program itself. And if bit 7 of the accumulator is 1, the zero flag is not set, the jump does not occur and the program moves on to the next instruction. This pops the contents of the stack back, restoring the AF registers to their former values and returns to the BASIC interpreter.

Once it has been given the go-ahead, the program disables the interrupt with the **di** command—you don't want your interrupt routine being interrupted. Then the contents of all the other registers are pushed onto the stack.

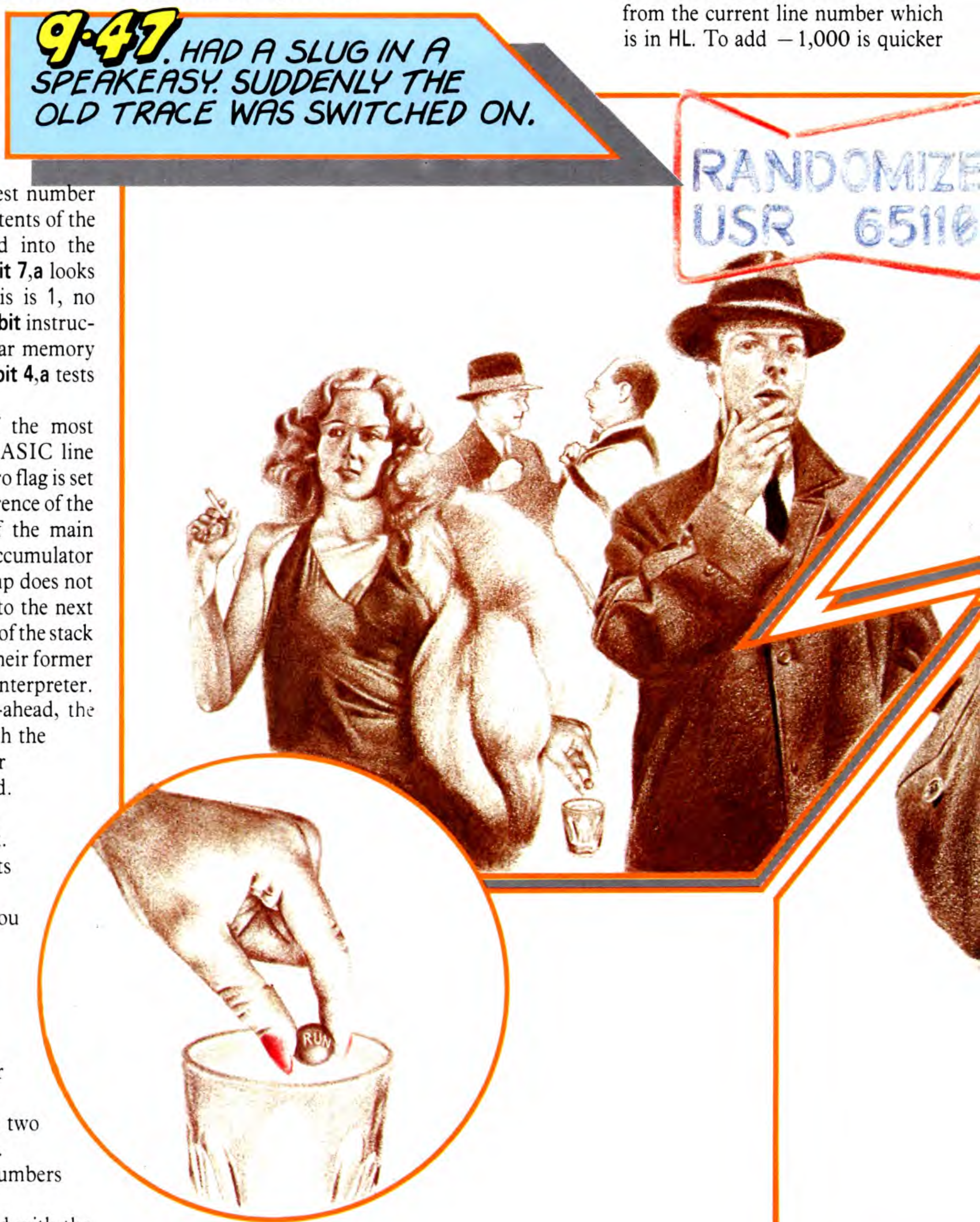
The instruction **ld de,20726** puts the address of the screen position immediately before the first one you are going to POKE a digit into, into the DE register. As the program has to refer back to this later, it is stored in the variable **posn** with **ld (posn),de**. There is no instruction to load a variable or a memory location with a number directly as you would need to give two pieces of data with the instruction. Loading memory locations with numbers must be done via a register.

The HL registers are then loaded with the contents of memory locations 23,621 and 23,622. These hold the number of the line of BASIC currectly being executed, remember. Everything is now set to call the first machine code subroutine labelled **lineno**.

The BASIC line number now in the HL registers is in hex, so it has to be converted into decimal before it is printed on the screen. To do this it starts by working out the fourth digit, in other words the thousands. And to do that you subtract 1,000 from the number until it won't go any more and count the number of times you do it.

So first of all you load the BC registers with −1000 and call another subroutine, **prt**, to do the repeated subtraction. In **prt**, the first thing the microprocessor does is **xor a**—that is exclusive or (see page 287). In machine code an **xor** always works on the A register, so **xor a** exclusively ors the contents of the A register with itself, so the result must be 0. This is simply a quick way to put 0 in the A register—**ld a,0** takes one byte more.

The **add hl,bc** subtracts 1,000 from the current line number which is in HL. To add −1,000 is quicker



and easier than subtracting 1,000 as you don't have to bother about checking the carry flag. The A register is then incremented and acts as a counter. Then **jr c,prtlp** checks the carry flag and makes a relative jump if it is set.

If you consider what the BC registers actually contain you will understand how this works. As you have seen on page 180, negative numbers are represented in computers as very large positive numbers of restricted length. The BC register pair holds 16 binary ones, minus 1,000 in decimal. So when you add any number greater than 1,000 to it, the register pair overflows and sets the carry flag.

When the carry flag is set, the jump is made, another 1,000 is subtracted, the counter is incremented and the whole process repeats until the contents of the HL are less than 1,000 and the addition does not set the carry flag.

The problem is that when the microprocessor comes out of the loop, the subtraction has been made and the incrementation has been performed one too many times. So the first thing that has to be done is to add 1,000 to (or subtract −1,000 from) the HL register and decrement the A register.

The A register now contains the value of the thousands digit. By adding 48, you get the ASCII code of the character. The contents of the HL are what is left over, and they are saved by pushing them onto the stack—so they can be recalled when you want to work out the hundreds, the tens and the ones. Then yet another subroutine, **print**, is called.

Print's first instruction, **ld bc,(23606)** loads the BC registers with the contents of memory locations 23,606 and 23,607 which is system variable pointing to the character set. The H register is then set to nought and the L register is loaded with the contents of A, the ASCII code of the character you want to print. The HL register is then used as a 16-bit accumulator.



To print out a character on the screen you first have to locate it in the character set. And as each character is made up of eight bytes, you have to count the ASCII value times eight along from the beginning to find the character you want.

ADD DON'T MULTIPLY

But multiplication is a tortuous business in machine code. There is no instruction to do it and it must be performed by shifting the binary digits about. Addition is easier, so to multiply the ASCII value by eight, you simply double it—by adding it to itself—three times. Note that this isn't done in the A register because it is bound to overflow an eight-bit register—$48 \times 8 = 384$, the start of the character Ø, is greater than 255, the maximum capacity of an eight-bit register—but it won't overflow a 16-bit register.

The result is then added to the contents of the BC register to give the location of the first byte of the character required. The contents of the HL and DE registers are then exchanged so that the HL register can be used again.

The **ret** returns the microprocessor from the subroutine and **ld hl,posn** loads the screen position stored in **posn** into the HL register. Then the contents of **posn** are incremented by the indirect instruction **inc (hl)**. Be careful when using this instruction. It only increments the low byte. If it overflows, the carry flag will be set, but the high byte will not be incremented automatically. In this case there is no danger though. The low-byte of a screen location only reaches 255 at the end of a line. The **inc (hl)** instruction is very useful because it is the only command that increments the contents of a memory location. The others only increment the contents of registers.

The contents of **posn**—the screen location for the first digit—themselves are then loaded into HL. And the actual print routine, called **prtout**, is called.

The first instruction in the printout subroutine is **ld b,8**, which loads the B register with eight. This is going to be used as a counter while the eight bytes that make up the digit are POKEd onto the screen. The accumulator is then loaded with the contents of the memory location whose address is in DE—in other words, with the first byte of the appropriate digit. And the contents of the accumulator is then loaded into the memory location whose address is given by the contents of HL—in other words, the appropriate screen location. You cannot load the contents of one memory location directly into another without putting them into a register on the way, except with a block load command

that takes several bytes to set up. There is no **ld (hl),(de)** instruction for example. It takes two instructions to perform this switch which actually prints the top line of the pixels of the digit on the screen.

The contents of the H register are then incremented. As H contains the high byte of the HL register pair, this effectively increases the contents of HL by 256, so that it contains the address of the next pixel line of the digit.

Incrementing the DE registers moves this pointer along to the next byte of the digit in the character set and **djnz loop** sends the microprocessor back round the printout routine to fill in the next line of pixels. This loop is executed eight times, each time incrementing H—to move down the pixel lines on the screen—and DE—to move it along to the next byte of the digit in the character set. At the same time, the **djnz** instruction—decrement and jump if not zero—decrements the contents of the B register. So when the loop has been performed eight times—and the eight bytes comprising the digit are printed one below the next on the screen—the entire character has been printed, the B register contains Ø, the non-zero condition of the **djnz** is not fulfilled and the microprocessor moves onto the next instruction. And the **ret** returns it to where the subroutine was called.

The **pop hl** pulls the contents of the top two memory locations off the stack and puts them into the HL registers. Looking back, you will see that the last thing pushed onto the stack were the contents of the HL register when it contained the remainder after the thousands digit was worked out. The **ret** then sends the processor back to the line that reads **statno ld bc,−1ØØ** and the process is repeated to work out the hundreds digit. When that has been printed on the screen next to the thousands, the tens—and then the ones—are worked out in the same way and printed next to them.

That done, the processor returns to where the **lineno** subroutine was called. The variable **posn** is then loaded with 20,731—the address of the screen location character spaces to the right of where the line number printing routine started. This leaves room for four digits containing the line number and one space.

The HL register is then loaded with the contents of 23,623 and 23,624, which is the system variable that contains the number of the statement in a line of BASIC that is currently being executed. But the maximum number of statements that can be included in a line is 128 on the Spectrum. So the high byte—which is in fact the value of another system variable—must be set to nought with the instruction **ld h,Ø**. The routine that works out the decimal characters and pokes them on the screen is then executed again. But this time it is called at the label **statno** as only three digits are needed.

When the statement number has been poked onto the screen the processor returns to the main program again to execute a small routine which makes the digits appear white on a small black panel. It does not matter that this is done after the numbers themselves have been printed. The attributes file—which contains the colour information—and the display file—which specifies the contents of the character spaces—are completely independent and are only superimposed on each other by the TV screen. Besides, the whole routine takes less than a 50th of a second—the rate at which the TV screen is scanned—so there is no chance of you seeing a digit printed black out of white then suddenly reverse to white out of black.

The **ld hl,23286** is the address in the attributes file of the character square before the first digit. This is to give the numbers a border. The number 71—which produces black paper and bright white ink —is loaded into that location. DE is then loaded with the address of the next location and the BC register is loaded with 9. The BC register is going to be used as a counter.

The block load instruction **ldir** loads the contents of the location pointed to by HL into the location pointed to by DE, increments HL and DE, decrements B and checks that it does not, now, contain Ø. If not, it repeats. In other words, it copies the black paper, bright white ink attribute from the first character square along the next nine.

THE PAUSE FACILITY

The next five instructions give you a pause facility so that you can halt the trace—and the program—at any point. The **in** instruction looks at the various input ports. The particular port you want to look at is the keyboard port which is number 254. And the part of it you want to examine is the right-hand end of the bottom row of keys—from B to BREAK/SPACE. So **ld a,127** loads the accumulator with 127. This is then used as a parameter by the **in** instruction. The 254 is data that is actually written into the program. Altogether **ld a,127** and **in a,(254)** loads the value of the right-hand end of the bottom keyboard line into the accumulator. (Most commercial assemblers require brackets round the port number, the *INPUT* assembler does not.)

From B to BREAK/SPACE there are only five keys, and whether they are pressed is specified by one bit each. That leaves three spare bits. ORing the accumulator with 224 makes the three most significant bits—which are the spare ones—1 and leaves the rest alone. 224 is 111ØØØØØ in binary. See pages 286 and 287 for more information on using OR with binary.

When a key is not being pressed, the value it gives is 1. When it is being pressed, its value changes to Ø. To stop the trace, two keys have to be pressed simultaneously, the SYMBOL SHIFT and the BREAK/SPACE. This has been done so that the keyboard can be used normally to edit the program while the trace is on.

It's a fail-safe device really, because when a BASIC program has finished RUNning the number of the last line executed is left in the system variable until another direct command is entered. So this way the trace is not turned off immediately—and if a single key press was used to make it pause, confusion could arise.



Pressing BREAK/SPACE changes the zero bit from 1 to Ø and SYMBOL SHIFT does the same to bit one. If none of the keys are pressed and the three spare bits are masked to 1s, the output of this part of the keyboard will be 255 or 11111111, that is, the byte will be full of 1s. But if BREAK/SPACE and SYMBOL SHIFT are pressed the output will be 252—that's 111111ØØ.

So when the input from port 254 is loaded into the accumulator, it is compared with 252 by **cp 252**. If the accumulator does contain 252, the zero flag is set. And **jr z**—jump relative if zero—loops the processor back round the routine. So, if the BREAK/SPACE and the SYMBOL SHIFT keys continue to be pressed it will go round and round. And as this is an interrupt routine, the main program will be halted too.

When no keys are pressed, the processor will move straight on to **pop ix, pop hl, pop de, pop bc** and **pop af** which restore the registers to their values at the beginning of the interrupt by pulling the stored values one after the other back off the stack.

The **ei** instruction enables the interrupts again and **ret** returns the program to BASIC. The BASIC program will then RUN until the next interrupt.

The only thing in the assembly language listing that has not been mentioned is the last instruction: **posn defw Ø**. This is not part of the program and when it is translated into machine code it gives ØØØØ—in other words, nothing at all.

This last instruction is an *assembler directive*. And what it does is to assign two bytes to hold data. The data held in there, naturally, is the current value of **posn**.

The instruction **defw** means define word and sets aside two bytes for data. (A similar instruction—**defb** or define byte—sets aside one byte.) Here, the Ø fills the two bytes with Ø. A 1 would put Ø1 in the low byte and ØØ in the high byte. You can define a word or a byte with any value you like in it, provided that value will fit into the two bytes or the one byte respectively.

Be very careful when using **def** instructions. They must not be put in any position where they might be executed by the processor. Otherwise they will corrupt your program. Here, this **defw** is safe because it is after a **ret** at the end of the program.

TRACKING DOWN ERRORS

To use your trace, first enter the BASIC program to be checked. Then CLEAR down to 65,109 and enter the trace.

Before you run the trace, make sure that you have SAVEd it on tape. If your assembler is working, you can use the normal save command. This will save the source code—that is, the assembly language—along with assembler itself. When you have got the trace working it is best to save the machine code itself, so you don't have to re-assemble it each time you want to use the trace. Do this by using the instruction:

```
SAVE "TRACE" CODE 6511Ø,176
```

To LOAD it back into the machine first CLEAR to 65109 then key in:

```
LOAD "" CODE
```

If your assembler is not working, use the save option on your machine code monitor.

Once the trace is back in position in your Spectrum, call it with:

```
RANDOMIZE USR 6511Ø
```

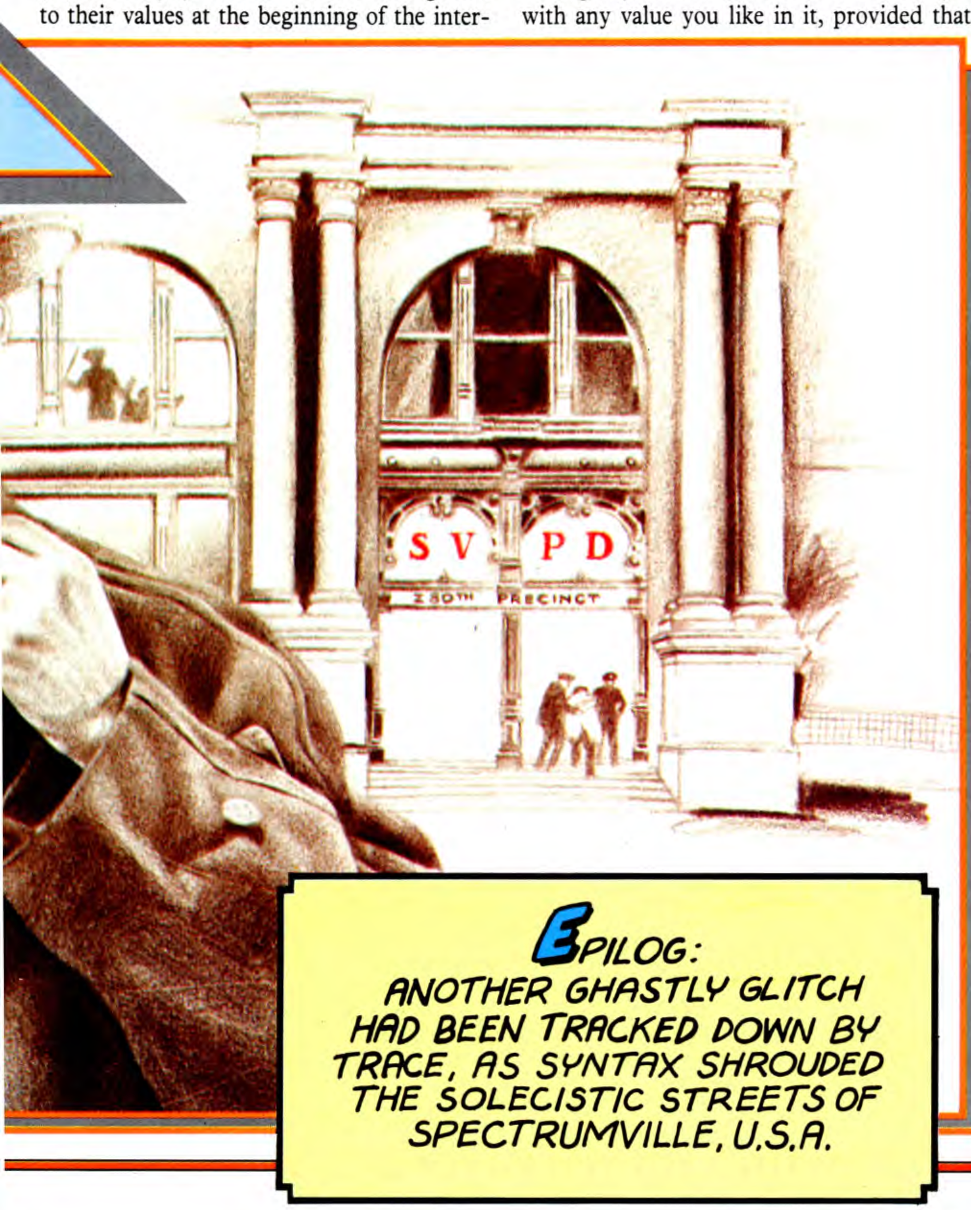
Nothing will happen until you RUN your BASIC program. Then the line number and statement number of the part of the program being executed will appear.

One of the easiest bugs to spot with a trace is an infinite loop. If you see the trace going over and over the same numbers you will know that you have got something wrong in a GOTO statement. Other things to watch for are sudden jumps to another part of the program, conditional IF ... THENs set to the wrong condition and lines not being executed at all.

When you run the trace, keep a copy of the BASIC program you are checking near at hand. You will want to check off each line as it is executed to see how the program is structured. At points where it runs into difficulty, step through the program slowly statement by statement, pausing between each by pressing the BREAK/SPACE and SYMBOL SHIFT key together.

When you have finished tracing the program, switch the trace off by calling the 'off' routine with:

```
RANDOMIZE USR 6512Ø
```



PICTURES FROM UDGS

Once you've defined a set of characters it's an easy matter to create a whole variety of pictures simply by rearranging the UDGs and changing the background

The last article in this series showed you how you can get around any limits your computer may have over the number of UDGs you can create—and how you can redefine the character set. This article is the first of two which show how you can save yourself a great deal of time and effort by using UDGs to form a screen picture.

WHY USE UDGs?

Suppose that you want to create a jungle picture—perhaps a title page for an exciting new game. There are basically two ways to do this. You could DRAW each part of the picture and colour it as required. But suppose you wanted a forest as part of your background. You would need to draw a trunk and tree-top for every tree that you have in your picture—a tedious business. And, of course, each of these trunks and tree-tops would be almost the same. If you could design a UDG, or several UDGs, which combine to form a tree, you could just PRINT the newly created characters wherever you want a tree (or, of course, PUT them if you have a Dragon or Tandy computer).

This has the obvious advantage that it saves you having to specify all the detail every time that you want to have a tree on the screen. It also has a number of other additional benefits.

One of these is that it saves time. Computers PRINT characters much faster than they DRAW high resolution lines (this is true even if the characters are user defined and very complicated). So, if you store your picture in UDGs, when the computer comes to the bit of your program which displays the picture, you don't have to wait around for a long time while the computer painstakingly brings your design to the screen: it appears almost instantly.

FREEDOM OF CHOICE

Another advantage is that you can often vary the size of the object very easily. With the example above, you could change the height of the tree simply by adding or subtracting one or more of the 'trunk' UDGs, or alter its foliage by using a different combination of 'leaf' UDGs. Of course, you could also do this with your computer's high resolution graphics commands, but UDGs are much more convenient. For a start, you are dealing in much more manageable numbers on the low resolution screen.

Once you have defined the UDGs for your picture, they remain in memory (unless you change them) although they are not necessarily permanently accessible for the keyboard: if you have several banks, or sets, of user defined characters in memory at once, then you have to 'switch in' the bank that is accessible from the keyboard. But this is simply a matter of altering the UDG pointer (the character set pointer on the Commodore 64 and Vic 20) when you want to 'switch in' the next bank of UDGs. (This is not applicable on the Acorn, where you use VDU commands to access all of the UDGs.)

This means that you can use the characters that you have designed as many times as you like within your programs: the only limit is the computer's memory. So it is as easy to have a forest in your program's picture, as it is to have just one tree.

Using UDGs does have slight drawbacks, though. To start with, you have to define all the characters, and type the DATA into your computer—although this is usually no worse than high resolution graphics. But you also have to use part of your computer's limited memory to store the UDGs as well, and unless you are careful you end up storing them twice. The next article in this series will show you how to avoid this repetition of memory.

WHICH PICTURES USE UDGs?

For this reason, there are some types of picture which you can draw much more efficiently with UDGs than others.

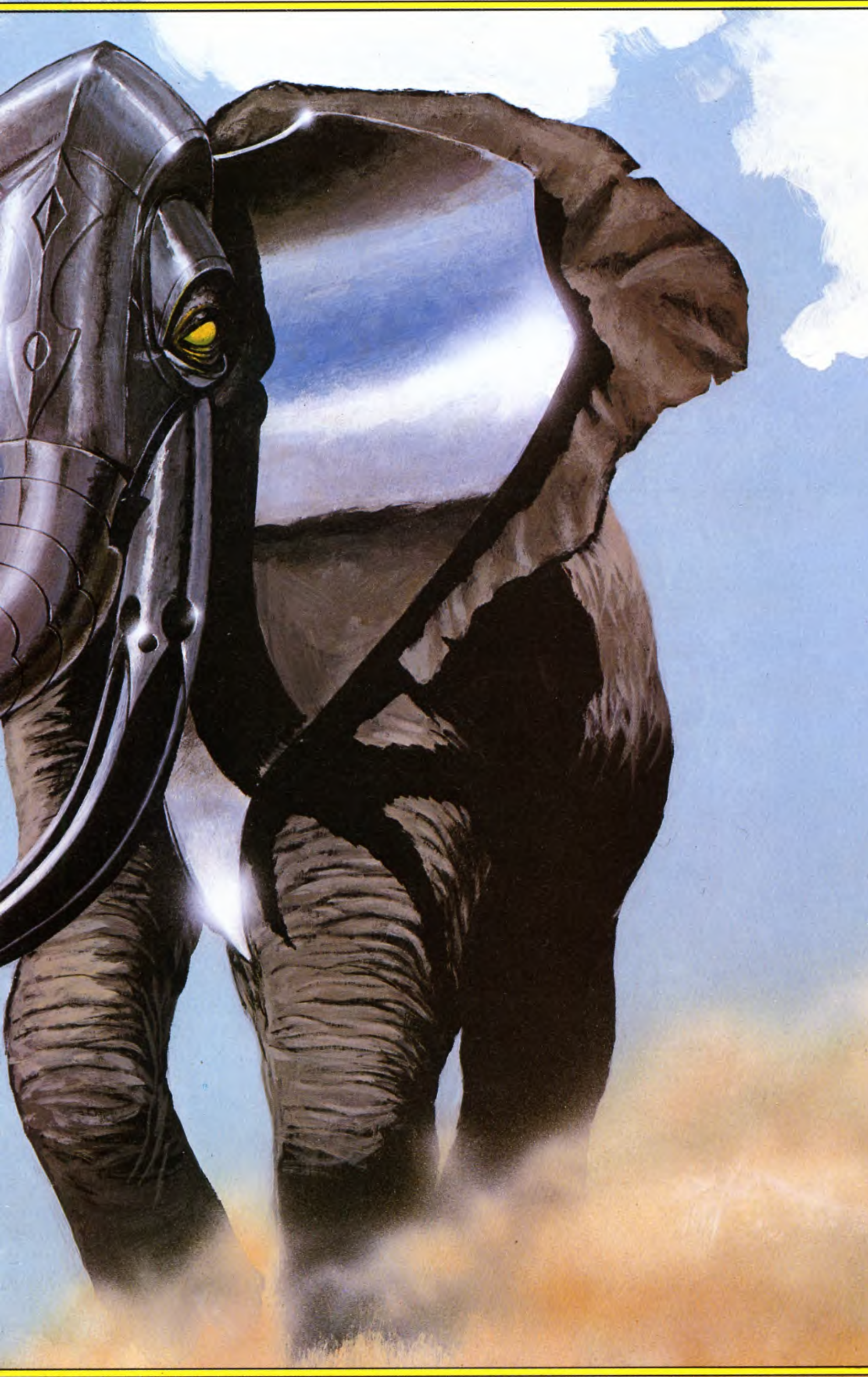
As a general rule of thumb, if your picture has a fairly small number of solid objects in sections that occur in a similar form several times, or if you wish to use an image a number of times during the program, then you can save yourself time and effort by using UDGs.

If, on the other hand, you want to have a highly detailed outline picture which you want to use just once, in the program, you would probably be better off using your computer's high resolution graphics commands.

For example, your picture might feature a brick wall. You could construct it quite



- HOW TO DECIDE WHEN TO USE UDGS
- BUILDING UP A PICTURE FROM SEVERAL CHARACTERS
- MAKING A JUNGLE PICTURE
- CREATING A CROCODILE, ELEPHANT AND TREES
- DRAWING THE BACKGROUND
- MIXING HIGH AND LOW RESOLUTION GRAPHICS



simply using one or two UDGs which you PRINT or PUT repeatedly over the area of the screen where you want to display the wall. The alternative would be for you to draw a series of lines across a coloured area to simulate the joins between each brick.

Another example would be the jungle scene mentioned above. You have already seen that you can save time by defining UDGs for the trees that any jungle scene must contain. The animals and so on which would accompany the trees are also very well suited to being defined as UDGs: you might want to use them elsewhere in the program, or to animate them, or to have large numbers of them.

A JUNGLE PICTURE

Here are some programs which define typical characters you would be likely to include in the jungle scene mentioned above.

The DATA for the programs for the Spectrum, Commodore and Acorn computers is the same. It is listed from Line 13ØØ onwards, and these lines are printed immediately after the Acorn program.

On the Acorns, enter this command followed by NEW before you type in the program:

```
PAGE=PAGE+&6ØØ
```

This changes the start address of BASIC to protect the UDGs (see the last article).

S

```
1Ø CLEAR 635ØØ
1ØØ REM poke crocodile data
11Ø POKE 23676,255
12Ø FOR n=USR "a" TO USR "r"+7: READ
   a: POKE n,a: NEXT n
```



```
250 REM poke elephant data
260 POKE 23676,249
270 FOR n=USR "a" TO USR "m"+7:
    READ a: POKE n,a: NEXT n
280 REM poke tree data
290 POKE 23676,248
300 FOR n=USR "a" TO USR "o"+7:
    READ a: POKE n,a: NEXT n
400 REM background
410 BORDER 1: PAPER 8: CLS
420 FOR n=1 TO 8: PRINT PAPER 5;"□";
    TAB 31;"□": NEXT n
430 FOR n=1 TO 14: PRINT PAPER
    4;"□";TAB 31;"□": NEXT n
440 PLOT 0,110: DRAW 142,-100,-PI/3:
    PLOT 160,110: DRAW -60,-42,PI/3
450 REM print crocodile
460 POKE 23676,255: INK 2
470 PRINT AT 19,20;: FOR n=144 TO 155:
    PRINT CHR$ n;: NEXT n
480 PRINT AT 20,20;CHR$ 156;CHR$
    157;CHR$ 158;CHR$ 159;CHR$ 159;CHR$
    159;CHR$ 159;CHR$ 159;CHR$ 160;CHR$
    159;CHR$ 159;CHR$ 159
490 FOR n=0 TO 31: PRINT INK 1; PAPER
    4;CHR$ 161;: NEXT n
750 REM print elephant
760 INK 7
770 POKE 23676,249
780 PRINT AT 8,9;CHR$ 144;CHR$ 145;CHR$
    146;CHR$ 147;AT 9,9;CHR$ 148;CHR$
    149;CHR$ 150;CHR$ 151;CHR$ 152;AT
    10,10;CHR$ 153;CHR$ 154;CHR$
    155;CHR$ 156
790 REM print trees
800 POKE 23676,248
810 LET x=6: LET y=14: GOSUB 840: LET
    x=5: LET y=18: GOSUB 840
820 LET x=4: LET y=0: GOSUB 845: LET
    x=4: LET y=3: GOSUB 845
830 GOTO 850
840 PRINT INK 4;AT x,y;CHR$ 151;CHR$
    152;CHR$ 153;CHR$ 154; INK 2;AT
    x+1,y+1;CHR$ 155;CHR$ 156;AT
    x+2,y+1;CHR$ 157;AT x+3,y+1;CHR$
    157;AT x+4,y+1;CHR$ 157;AT
    x+5,y+1;CHR$ 158: RETURN
845 PRINT INK 4;AT x,y;CHR$ 144;CHR$
    145;AT x+1,y;CHR$ 146;CHR$ 147;AT
    x+2,y;CHR$ 148;CHR$ 149; INK 2;AT
    x+3,y+1;CHR$ 150;AT x+4,y+1;CHR$
    150;AT x+5,y+1;CHR$ 150: RETURN
970 INK 0
```

```
10 POKE 52,48:POKE 56,48:CLR
20 POKE 56334,0:POKE 1,35
30 FOR Z=0 TO 1024:POKE 12288+Z,
   PEEK(53248+Z):NEXT Z
40 POKE 1,39:POKE 56334,1
50 FOR Z=0 TO 143:READ X:
   POKE 13312+Z,X:NEXT Z
52 FOR Z=760 TO 983: READ X:
   POKE 13312+Z,X:NEXT Z
55 C$=CHR$(13):POKE 53272,28:
   POKE 53281,0:POKE 53280,6
60 PRINT "[graphics]":FOR Z=1 TO 30:
   X=INT(RND(1)*320)
70 POKE 1024+X,46:POKE 55296+X,
   RND(1)*6+2:NEXT Z
80 T$="[graphics]":
   C(0)=3
85 TT$="[graphics]":
   C(1)=6
87 PRINT"[graphics]":
   FORZ=1TO280:PRINT"[graphics]";:
   NEXT
90 PRINT"[graphics]"TAB
   (13)TT$"[graphics]"
   T$"[graphics]"T$TT$
95 PRINT"[graphics]"TAB(30)
   T$"[graphics]"T$"[graphics]"T$TT$
96 PRINT"[graphics]
   [graphics]"SPC(30)"[graphics]"CHR$(160)
   "[graphics]"
150 FORZ=0TO159:POKE1864+Z,102:
    POKE56136+Z,C(RND(1)*2):
    NEXTZ
160 PRINT"[graphics]"
    TAB(28);
162 IF RND(1)>.50 THEN PRINT
    "[graphics]π@ABCDEFGHIJK";SPC(28);
    "[graphics]LMNOOOOOPOOO[graphics]";:
    GOTO170
164 PRINT"π□□□□□□□□□□□
    □□";SPC(28);"[graphics]@ABCDEFGHIJK[graphics]";
170 FOR D=1 TO 200:NEXT D
190 GOTO 160
```

```
10 MODE 1
20 VDU 23;8202;0;0;0;
30 VDU 19,0,4,0,0,0,19,2,2,0,0,0,19,
   3,3,0,0,0
40 *FX20,6
50 FOR T=128 TO 173
70 VDU 23,T
80 FOR P=1 TO 8
```



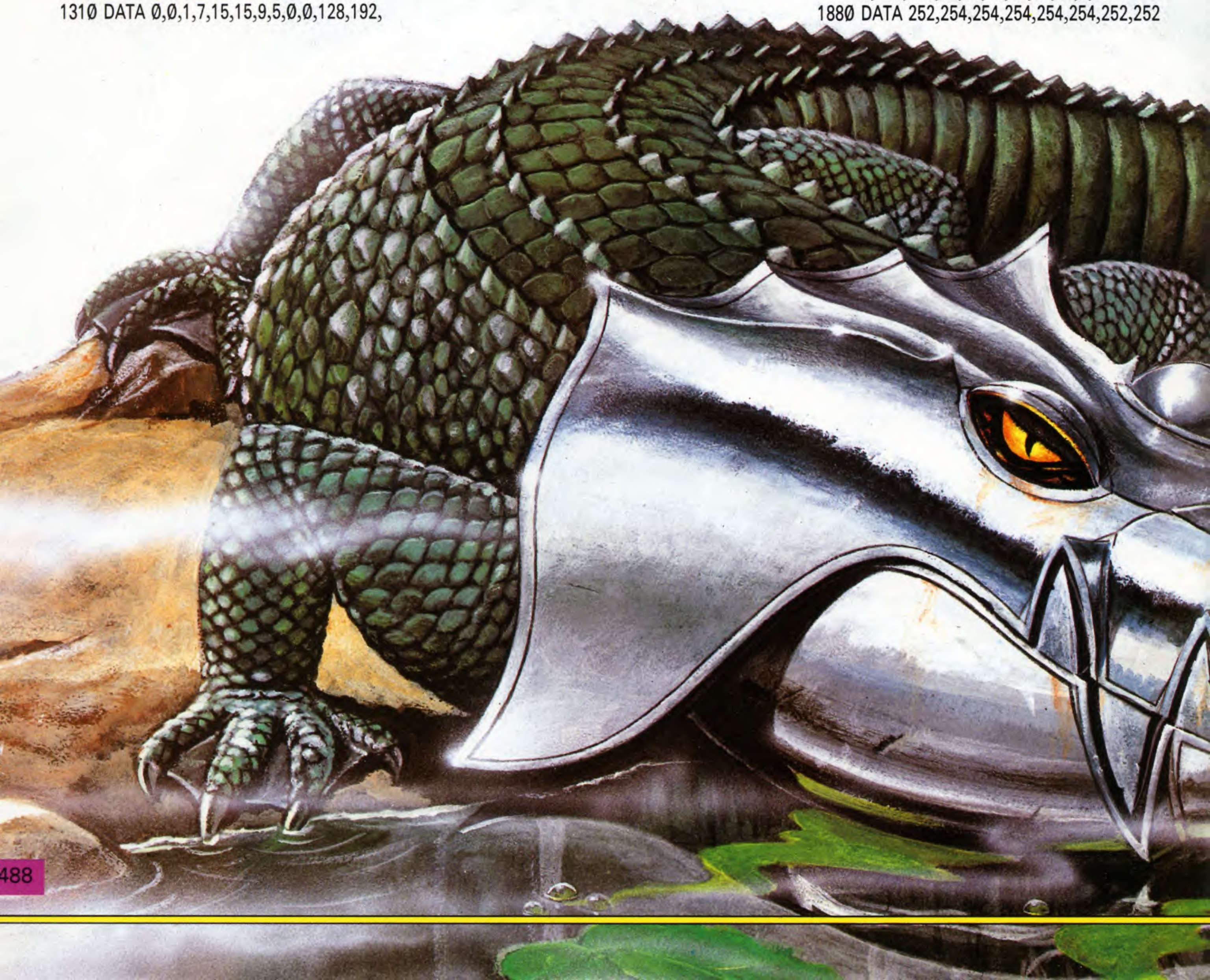
```
90 READ A
100 VDU A
110 NEXT
120 NEXT
160 PROCLAND
170 PROCALI(896,96,128)
200 PROCELE(800,460,146)
260 VDU 30
270 GOTO 270
280 REM
290 DEF PROCALI(X,Y,Z)
300 VDU 5
310 GCOLØ,1
320 MOVE X,Y
330 FOR T=Z TO Z+14
340 VDU T
350 IF T=Z+11 THEN MOVE X,Y-32
360 NEXT
370 FOR T=1 TO 5:VDU Z+15:
    NEXT
380 VDU Z+16:VDU Z+15,Z+15,
    Z+15
390 VDU 4
400 ENDPROC
640 DEF PROCELE(X,Y,Z)
650 VDU 5
660 GCOLØ,3
670 MOVE X,Y:VDU Z,Z+1,Z+2,
    Z+3
680 MOVE X,Y-32:VDU Z+4,Z+5,
    Z+6,Z+7,Z+8
690 MOVE X+32,Y-64:VDU Z+9,
    Z+1Ø,Z+11,Z+12
700 VDU 4
710 ENDPROC
720 DEF PROCTREE1(X,Y,Z)
730 VDU 5
740 GCOLØ,2
750 MOVE X,Y:VDU Z,Z+1
760 MOVE X,Y-32:VDU Z+2,Z+3
770 MOVE X,Y-64:VDU Z+4,Z+5
780 GCOLØ,1
790 MOVE X+32,Y-96:VDU Z+6
800 VDU 1Ø,8,Z+6,1Ø,8,Z+6
810 VDU 4
820 ENDPROC
830 DEF PROCTREE2(X,Y,Z)
840 VDU 5
850 GCOLØ,2
860 MOVE X,Y:VDU Z,Z+1,Z+2, Z+3
870 GCOLØ,1
880 MOVE X+32,Y-32:VDU Z+4,
    Z+5
890 VDU 1Ø,8,8,Z+6,1Ø,8,Z+6,
    1Ø,8,Z+7
900 VDU 4
910 ENDPROC
920 DEF PROCARC(X1,Y1,X2,Y2,A,B)
930 LOCAL C,D
940 R=SQR((X1-A)↑2+(Y1-B)↑2):
    R2=SQR((X2-A)↑2+(Y2-B)↑2)
950 C=ASN((X1-A)/R):
    D=ASN((X2-A)/R2)
960 MOVE A+R*SIN(C),B+R2*COS(C)
970 FOR T=C TO D STEP .Ø5
980 DRAW A+R*SIN(T),B+R2*COS(T)
990 NEXT
1000 ENDPROC
1010 DEF PROCLAND
1020 Z=145:VDU31,27,31:COLOURØ:
     COLOUR131:FOR T=1 TO 12:VDU Z:NEXT
1030 PROCARC(Ø,800,400,Ø,Ø,-200)
```

```
1040 PROCARC(600,240,1300,500,
   1280,0)
1050 GCOL0,2:FOR T=0 TO 246:
   PLOT77,0,T:NEXT
1060 FOR T=32 TO 500:PLOT77,1276,T:
   NEXT
1070 P=630
1080 FOR T=1 TO 8
1090 PROCTREE2(850+RND(300),
   P+RND(20)-40,166)
1100 PROCTREE1(900+RND(250),
   P+RND(20),159)
1110 P=P-10
1120 NEXT
1130 FOR T=1 TO 20
1140 PROCTREE2(RND(500),
   300+RND(50),166)
1150 NEXT
1210 ENDPROC
```

```
1300 REM CROCODILE
1310 DATA 0,0,1,7,15,15,9,5,0,0,128,192,
   248,255,127,95,1,3,6,12
1320 DATA 62,255,255,255,192,224,176,159,
   191,255,255,255
1330 DATA 0,0,0,0,0,248,252,255,0,0,0,0,
   0,0,1,207,0,0,0,1,15,127,255,255
1340 DATA 0,3,63,255,255,255,255,255,127,
   255,255,255,255,254,249,247,248
1350 DATA 255,255,255,255,15,255,255
1360 DATA 0,224,254,255,255,255,255,255,
```



```
   0,0,0,192,248,255,255,255,0,2,4,7,7
1370 DATA 3,0,0,21,1,164,73,255,255,0,0
1380 DATA 255,127,63,63,255,255,127,31,
   255,255,255,255,255,255,255,255
1390 DATA 239,239,239,239,239,247,247,
   247,60,255,255,255,255,255,255,255
1780 REM ELEPHANT
1790 DATA 0,0,0,8,28,25,51,51,0,0,0,126,
   255,193,253,253,0,0,0,0,0,255,255
1800 DATA 255,0,0,0,0,0,248,252,254
1810 DATA 102,111,111,111,125,57,26,0,
   253,251,251,251,231,31,15,15,255,255
1820 DATA 255,255,255,255,255,255,254,
   255,255,255,255,255,255,255
1830 DATA 0,0,128,64,32,16,12,0,15,15,
   15,14,14,14,14,31,255,240,224,224
1840 DATA 224,224,224,224,255,63,59,59,
   57,57,121,248
1850 DATA 0,0,128,192,224,224,128,0
1860 REM TREE 1
1870 DATA 0,0,0,0,1,1,3,7,0,0,0,0,224,
   240,248,248,15,63,63,63,31,15,3,3
1880 DATA 252,254,254,254,254,254,252,252
```



```
1890 DATA 3,3,3,3,3,1,Ø,Ø,248,248,248,
   248,248,248,240,96,96,96,96,96,96,96
1900 DATA 96,96
1910 REM TREE 2
1920 DATA Ø,3,15,31,127,127,63,1,7,15,
   255,255,255,255,255,255,15,63,255,255
1930 DATA 255,255,255,255,Ø,128,248,248,
   248,248,240,224
1940 DATA 255,227,96,48,24,25,13,15,252,
   240,96,96,192,192,128,128,7,7,7,7,7
1950 DATA 7,7,7,7,7,7,15,15,15,31,63
```

```
10 CLEAR1ØØØ:CLS
20 DIMC(59),E(17),T1(1),T2(7),
   F1(7),F2(7)
30 PMODE3,1:PCLS
40 W$="L6UL6D":W$=W$+W$+
   W$+W$
50 DRAW"BM1,4C4RUR2UR2DR2DR4DR8UR
   4UR2UR2UR2UR4DR2DR2DR4D2R6DR2DR
   4DR4UR4UR4UR4UR4UR4UR4UR4UR
   1ØDR6DR4DR2DR2DR2D15"+W$+
   W$+"L6UL6DL4BU14DR11FRFR7FR
   3FR9FGL7GL21HL2D3FR21FR3F"
60 PAINT(5Ø,1Ø),4
70 DRAW"BM98,5C1L8GLGLGD5BFD2RFR
   3FBM32,2GFREHLBM1,8C2FRERFBR
   4UBM+4,2;RBR3RBM+3,1;
   RBM+3,1;RBR3RBL8BDLBL5NEBL
   6NEBL4EBL5E"
80 GET(Ø,Ø)-(112,2Ø),C,G
230 PCLS
240 DRAW"BM4,ØC2DG2DG2D3R2DE2UE
   2UE4R3FRFR11F3RFRFR3DBL8NU3D
   4F2DGH2UHL3D5LURU3HL6D5L2BU
   6L3D6NL2U6LH2LNGUHL2"
250 PAINT(2Ø,7),2
260 DRAW"BM12,3C3R2D4G2BM8,4R"
270 GET(Ø,Ø)-(37,17),E,G
280 PCLS4
290 DRAW"BM7,19C1H3U5H5UE6R2F3DF
   2D2G3D3G2D2":PAINT(7,7),1
300 DRAW"BM2Ø,5E2R2E2RFR4E2F2R2E
   2R5FRFDG3LGLGLGL5HL5H2L4":
   PAINT(34,5),1
310 GET(Ø,Ø)-(13,19),F1,G:
   GET(2Ø,Ø)-(49,9),F2,G
320 PCLS:DRAW"BMØ,ØC4D2ØBE2ØF
   3DFD15G2NL2R8HL4EU13E4U2"
330 GET(Ø,Ø)-(1,2Ø),T1,G:GET
   (2Ø,Ø)-(31,22),T2,G
340 PCLS3:SCREEN1,Ø
350 CIRCLE(255,191),16Ø,1,.6:
   PAINT(23Ø,18Ø),1
360 CIRCLE(Ø,191),14Ø,2,.35,.75,1:
   PAINT(1Ø,18Ø),1,2
380 PUT(2Ø6,1ØØ)-(243,117),E,PSET
390 FORK=1TO1Ø:READX,Y:PUT(X,Y)
   -(X+13,Y+19),F1,AND:PUT(X+6,
   Y+2Ø)-(X+7,Y+4Ø),T1,OR:
   NEXT
400 FORK=1TO1Ø:READX,Y:PUT(X,Y)
   -(X+29,Y+9),F2,AND:PUT(X+8,
   Y+9)-(X+19,Y+31),T2,OR:
   NEXT
410 COLOR2:LINE(138,187)-(255,
   187),PSET:PAINT(255,191),2:
   PAINT(255,191),3,1
420 PUT(143,166)-(255,186),C,PSET
450 DATA 16,11Ø,24,113,34,1Ø8,48,11Ø,
   56,1Ø8,19Ø,8Ø,198,82,212,84,21Ø,
   79,24Ø,7Ø
460 DATA 2,12Ø,18,122,28,116,46,118,
   6Ø,124,16Ø,9Ø,174,95,19Ø,9Ø,214,
   86,226,9Ø
470 GOTO47Ø
```



If you type in and RUN the program above for your computer, you can see the kind of picture that you can produce with UDGs. Don't worry if the top half of the screen looks a bit empty at the moment, as the next article in this series will finish off the picture.

You can see how versatile UDGs can be by looking at the water underneath the crocodile. This is actually just one UDG which is repeated for the length of the stretch of water. (The Dragon and Tandy program is slightly different here, in that the array in which the crocodile is stored includes this water: the water is not a separate UDG, as it is in the other programs.)

You will have noticed that the picture does not consist solely of UDGs: it also has some high resolution lines in it, which give the impression that there are hills in the picture.

While UDGs can have many advantages over high resolution graphics commands, you can often obtain very good results by using both in the same picture, as with this example.

The next article in this series will add more to this picture, so SAVE it on tape to avoid having to type it in again later.

HOW THE PROGRAM WORKS

If you don't understand how the programs produce the extra user defined graphics characters, you should look at the article on pages 450 to 457, which explains this.

The programs can all be split into two parts: the first half, which defines all the characters, and the second part which PRINTs (or, if you have a Dragon or a Tandy, PUTs) the UDGs in their correct positions.

The Spectrum program POKEs in the DATA

in sections, for each animal and the trees, as do the Commodore and Acorn. The Dragon and Tandy do not use DATA to store the information but DRAW the various images one by one, and then GET them into UDGs.

Once the UDGs have been defined, the programs go on to display the picture on the screen.

The Spectrum uses a series of PRINT ATs for this, PRINTing each animal and the trees one at a time. It also uses local colour commands to produce the colour picture that you see when you RUN the program.

A local colour command is one which only applies to the statement in which it is found in a program Line. The general PAPER colour is set to 8, which is transparent. This leaves the PAPER the same as it already is, to ensure that the background is not harmed by the UDGs which are PRINTed.

MULTICOLOURED BACKGROUND

The background is set up by Lines 41Ø to 44Ø. Of these, the first sets the colour, and the next two produce the cyan sky and the green ground. The different colours of PAPER are achieved by PRINTing spaces in the relevant PAPER colour, using two FOR ... NEXT loops.

The trees are PRINTed with subroutines (Lines 81Ø and 82Ø), as there are several of each type of tree in the picture. The variables x and y in these lines refer to the PRINT AT coordinates of the trees. You can easily add more trees to the picture, if you want to, by working out some more coordinates (the article on pages 433 to 439 explains how you can work out these coordinates) and adding some more 'GOSUB...'s to these two lines.

The Commodore program begins with several POKEs to read the ROM character set into a protected area of RAM (the article on pages 450 to 457 explains exactly what each POKE does).

It then continues in the same way as the other programs—by POKEing the DATA for each of the user defined characters into RAM, so that you can use the characters. This finishes at Line 52.

Line 55 sets up a variable—C$—to be equal to CHR$ (13) which is RETURN. This is used to move the PRINT position onto the next line on the screen during the actual printing of the picture. The Line also sets the mode, the background, and the border colours, with three POKEs.

The screen is cleared in Line 6Ø, and the computer continues by actually starting on the picture. First, it prints stars in random positions (unlike the other computers' pictures, the Commodore picture is 'jungle by night'). The stars are simply dots, printed in a randomly chosen colour, by the FOR ... NEXT loop in Lines 6Ø and 7Ø.

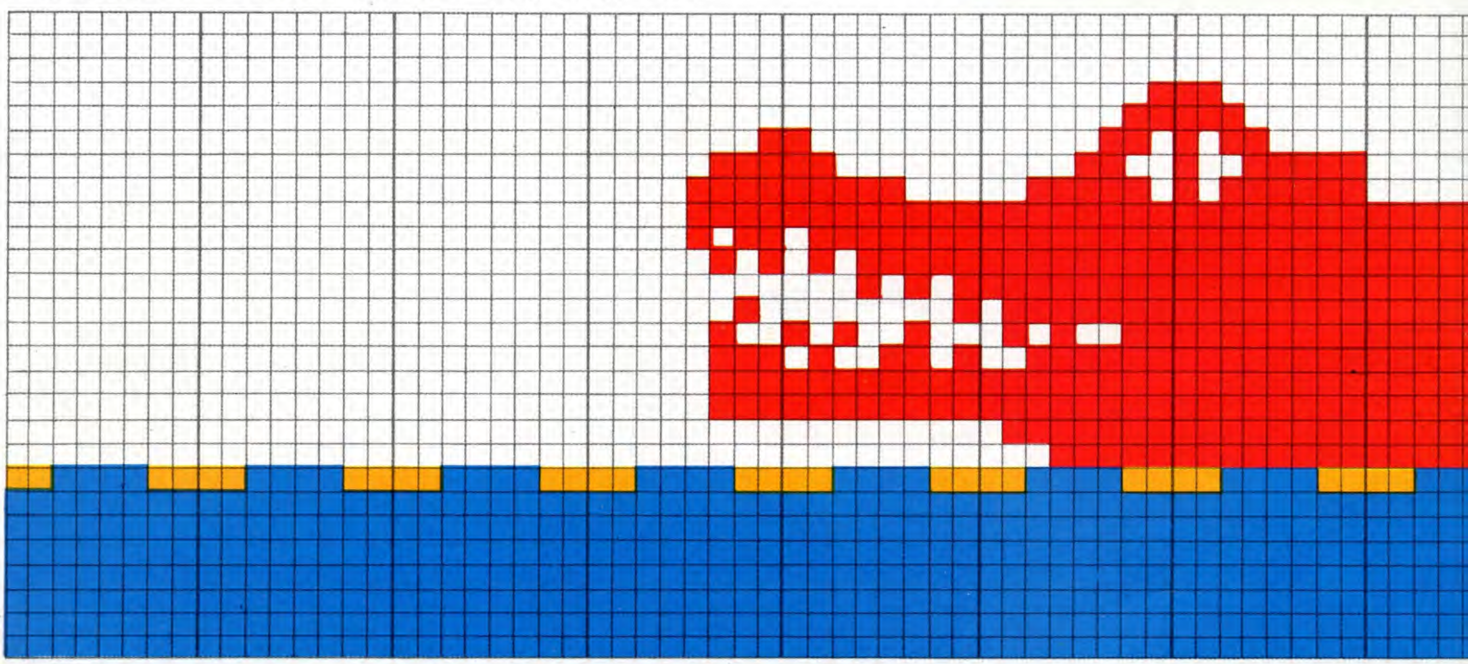
Next, the program actually PRINTs the UDGs in the positions determined by the cursor-positioning characters you can see inside each PRINT statement.

The trees are the first UDGs to be PRINTed, and, to make it easier, the characters which make them up are put into two string variables—t$ and tt$. Lines 9Ø and 95 then PRINT the two strings.

TROUBLESHOOTER

You should be very careful when typing in the DATA, as there is inevitably a lot of it.

- If your computer stops while it is RUNning the program with an OUT OF DATA error message it means that there is not enough DATA in your program.
- If you have told your computer to READ the correct amount of DATA, then there are two possible reasons for this: you might have missed one or more numbers altogether, or you might have typed full stops instead of commas. A full stop instead of a comma would change two numbers into one, as the full stop is taken to be a decimal point.
- The only solution to these two problems is to check, and recheck, your DATA until you find the message. It is helpful to add a PRINT command to the loop which POKEs in the DATA so that the computer PRINTs up each number as it POKEs it in. That way, you could check each number off from the screen as it is POKEd in.
- If you have typed in too much DATA, or the right amount but wrong numbers, the program should RUN without any problems, but you will see a rather strange set of pictures: the crocodile might have a trunk, for example, or the elephant's head might look surprisingly like a treetop. As before, the only solution is to check all the DATA entries.



The Line in between these, Line 87, adds some more background: the horizontal lines that you can see when you RUN the program. The other static creature, the elephant, is printed by Line 96.

The crocodile is printed with a slightly more complex routine so that it moves up and down in the water. This routine takes up lines 16Ø to 19Ø, and simply prints a line of spaces over the lower half of the crocodile if the number which is chosen randomly by Line 162 is greater than Ø.5.

If you do not like the moving crocodile, you can easily make him remain still by changing Line 19Ø to:

```
19Ø GOTO 19Ø
```

EXPLODING THE MEMORY

The Acorn program starts off by setting the MODE—(MODE 1)—setting the colours, and turning the cursor off. It then explodes the character set using the *FX 2Ø, 6 command (this command was explained in the article on pages 450 to 457) so that you can define more than the usual limit of 32 UDGs.

After this, Lines 5Ø to 12Ø actually define the graphics characters, using the DATA at the end of the program, as normal with the VDU 23, (character number followed by DATA), command.

Lines 16Ø, 17Ø, and 2ØØ call the PROCedures which put the UDGs onto the screen: PROCLAND is for the background and the trees, PROCALI is for the crocodile, and PROCELE for the elephant.

PRINTING THE UDGs

The PROCedures themselves use VDU commands to print the characters on the screen; where appropriate, this is within a FOR ... NEXT loop. PROCLAND does not just call the two tree routines, but also calls the PROCedure PROCARC, which draws the two arcs which form the hills of the background. Lines 1Ø5Ø and 1Ø6Ø fill in the two hills. The water on which the crocodile is resting/swimming is put in by Line 1Ø2Ø, also in PROCLAND.

The trees are placed randomly within certain limits, by the two PROCedures, PROCTREE1 and 2.

Once the computer has PRINTed all of this up on the screen, it sends the cursor to the top left hand corner of the screen, and enters a continuous loop to leave the picture intact (Line 27Ø). You can stop the program by pressing the [ESCAPE] key.

The Dragon and Tandy program begins by CLEARing some memory for the strings that it uses, and then DIMensions the arrays which store the UDGs. Except for the trees, there is only one UDG (and so array) for each character, as the arrays can be any size. The trees use two each, because half of each tree is red, while the other half is green.

DRAWING ANIMALS

The program continues by drawing each of the characters in turn (how this is done is explained on pages 191 and 192) and GETting them into the appropriate arrays. Line 8Ø GETs the crocodile, Line 27Ø GETs the elephant, the tree-tops are in Line 31Ø, and the tree-trunks are in Line 33Ø.

The next half of the program deals with actually combining all of the UDGs on the screen to form the picture. The first few Lines, starting with Line 34Ø, clear and set the correct screen, and put in the two hills which make up the background. The hills are coloured in using the PAINT command.

The two groups of trees, one on either hill, are PUT there by the FOR ... NEXT loops in Lines 39Ø and 4ØØ, using the DATA in Lines 45Ø and 46Ø to determine the position.

The elephant and crocodile are PUT onto the screen by Lines 38Ø and 42Ø, respectively. Line 47Ø is the last active Line of the program (if you ignore the two lines of DATA) and sends the computer into a continuous loop so that the picture remains intact on your screen until you press the [BREAK] key.

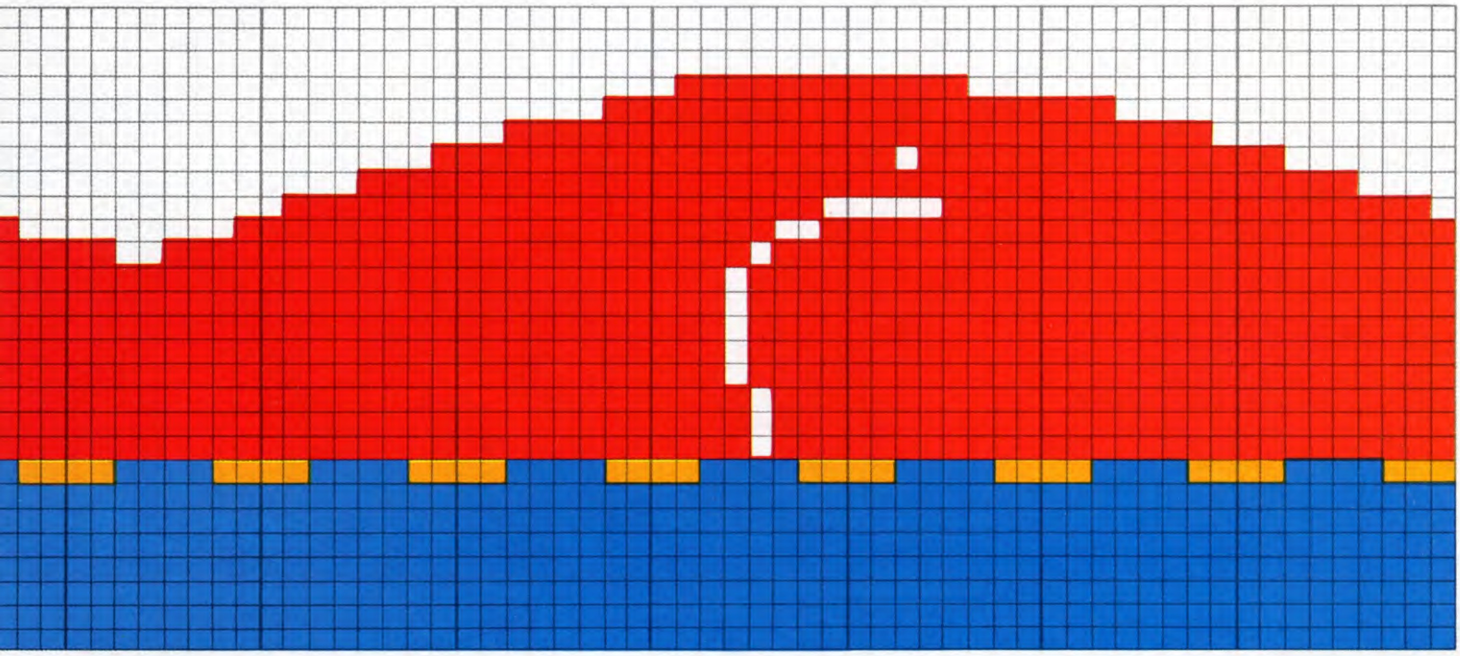
IN GENERAL

Now that you have these UDGs in memory, you can use them wherever you want to in your programs, and you can change the picture until it looks exactly how you think it ought to.

A HERD OF ELEPHANTS!

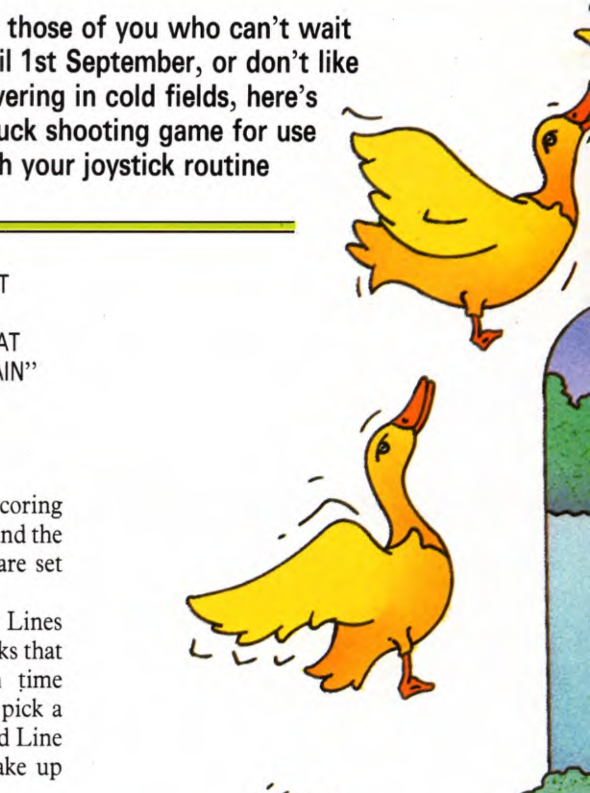
You might like to try replacing the single elephant that is there at the moment with a whole herd of them! You could do this in a similar way to the method used to display several trees on the screen: either with a FOR...NEXT loop, or with several GOSUBs.

In the next part of this feature you will see how to get some more characters into your scene—and how you can add some animation.



A DUCK SHOOTING GAME

For those of you who can't wait until 1st September, or don't like shivering in cold fields, here's a duck shooting game for use with your joystick routine



If you have followed through the first part of this article, you should have a program stored on tape which will allow you to move a gunsight around the screen. But although it's quite satisfying to see this kind of program written in BASIC, it hasn't got a lot of point as it stands.

The next step, then, is to use the new routine in a games program. Adding the program lines designed for your machine will give you a duck shooting game, although you could design your own graphic and shoot aircraft, water buffaloes, hot air balloons, or whatever takes your fancy.

The object of the game is to shoot ten ducks that appear for a short time at random positions on the screen. There's a score based on your accuracy. You get points for a hit—the quicker you are, the more points you get. And points are deducted for every miss.

LOAD the program back into the computer before you type in the additional lines.

Add the remaining lines to the joystick program to create the shooting and scoring facility:

```
120 LET hs=Ø
125 PRINT PAPER 2; INK 7; "□SCORE";TAB
    14;"HISCORE";TAB 31;"□"
135 PRINT PAPER 3; OVER Ø;AT Ø,23;hs
145 LET s=Ø
150 FOR n=1 TO 1Ø
160 LET dx=INT (RND*31)
170 LET dy=INT (RND*2Ø)+1
180 PRINT INK 6;AT dy,dx;CHR$ 144;CHR$
    145;AT dy+1,dx;CHR$ 146;CHR$ 147
190 POKE 23672,Ø: POKE 23673,Ø
205 PRINT OVER Ø; PAPER 3;AT Ø,8;s;"□"
210 IF IN 31<>16 THEN GOTO 4ØØ
220 IF ATTR (y,x)=14 AND ATTR
    (y+1,x+1)=14 THEN LET
    s=s+25Ø-(PEEK 23672+256*PEEK
    23673): BEEP .Ø2,3Ø: GOTO 41Ø
230 LET s=s-1Ø
400 IF PEEK 23672+256*PEEK 23673<2ØØ
    THEN GOTO 2ØØ
410 PRINT INK 7;AT dy,dx;CHR$ 144;CHR$
    145;AT dy+1,dx;CHR$ 146;CHR$ 147
420 NEXT n
430 IF s>hs THEN LET hs=s
440 PRINT OVER Ø; PAPER 3;AT Ø,8;s;AT
    Ø,23;hs
450 PRINT OVER Ø; PAPER 3; FLASH 1;AT
    1Ø,2;"PRESS ANY KEY TO PLAY AGAIN"
460 FOR n=1 TO 1ØØ: NEXT n
470 IF INKEY$="" THEN GOTO 47Ø
480 CLS : GOTO 125
```

Lines 12Ø, 125 and 135 look after the scoring additions. The high score is set to zero and the displays for the score and high score are set up.

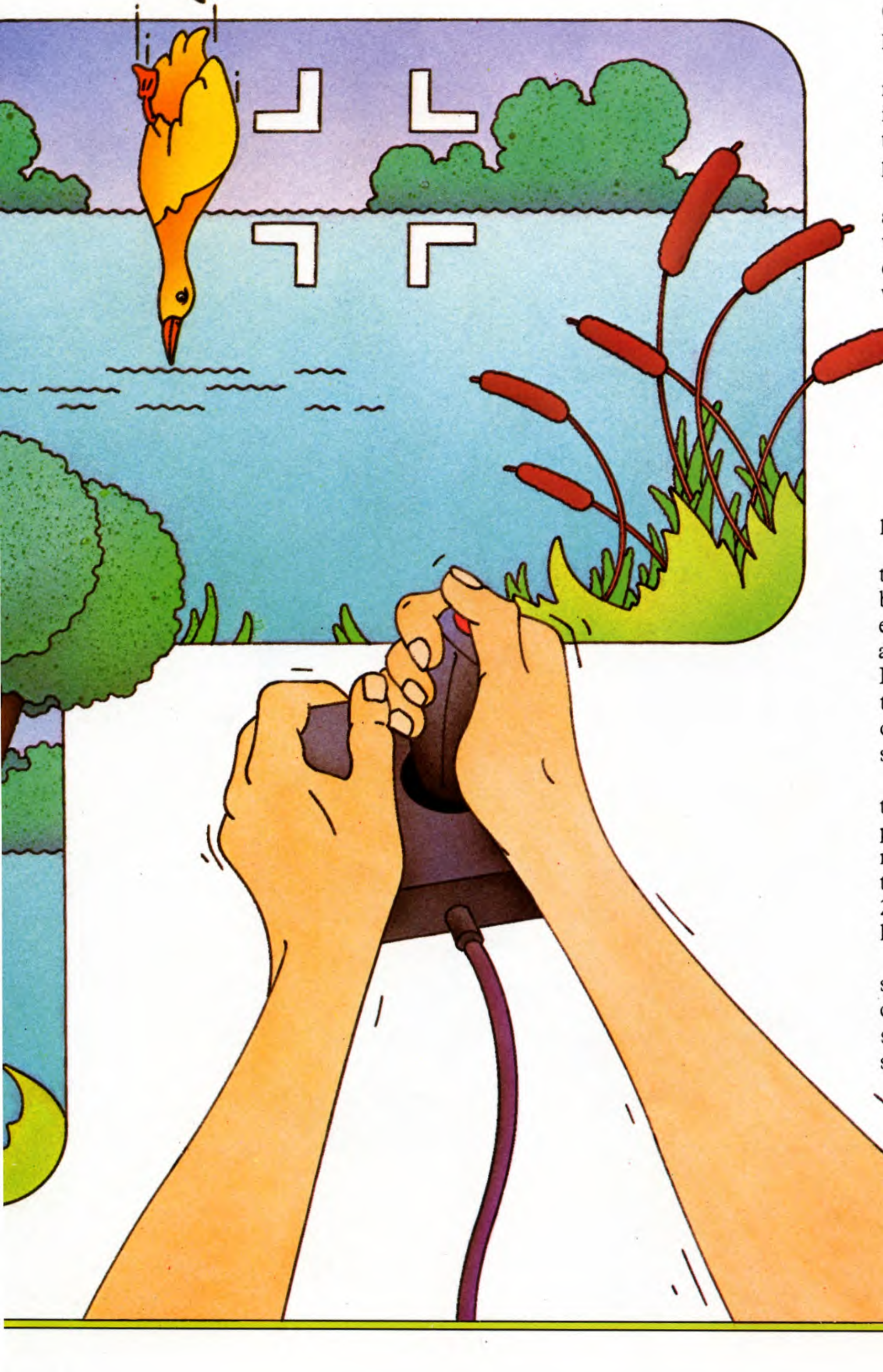
The FOR ... NEXT loop bounded by Lines 15Ø and 42Ø displays a series of ten ducks that the player attempts to shoot. Each time through the loop, Lines 16Ø and 17Ø pick a new position for the duck at random and Line 18Ø PRINTs the four UDGs which make up the graphic at that position.

The timer is initialized by Line 19Ø, and will be used to provide a time limit within which the duck has to be shot.





- USING THE JOYSTICK ROUTINE WITHIN A GAME
- DUCK GRAPHICS
- DISPLAYING THE DUCKS ON THE SCREEN
- A TIMING ROUTINE
- DETECTING THE FIRE BUTTON
- CHECKING FOR A HIT
- ADDING UP THE SCORE
- A HIGH SCORE ROUTINE



Once the time limit has been exceeded the program will print it somewhere else.

After RETURNing from the joystick subroutine, Line 2Ø5 blots out the first character in the score display—this is a precaution so that if the score goes down, none of the previous score will still appear on the screen.

The joystick is equipped with a fire button, and IN 31 has to be checked to detect presses. A value of 16 means that the fire button is being depressed. Line 21Ø detects firing, but only when the gunsight has stopped. It is possible to check if the player is firing when the gunsight is moving, but it would need eight more checks—one for each joystick direction. This isn't really feasible in BASIC because all these checks would slow the program down by a huge amount.

If the fire key has been pressed, the program then carries on to check if the duck

If the fire key has been pressed, the program then carries on to check if the duck has been hit by using ATTR in Line 22Ø. ATTR will be explained properly later, but broadly, it looks at the colour of a particular part of the screen. If the colour corresponds to that of the duck the program counts it as a hit and the score is calculated from the amount of time taken to shoot it. There's a BEEP to signal success, too.

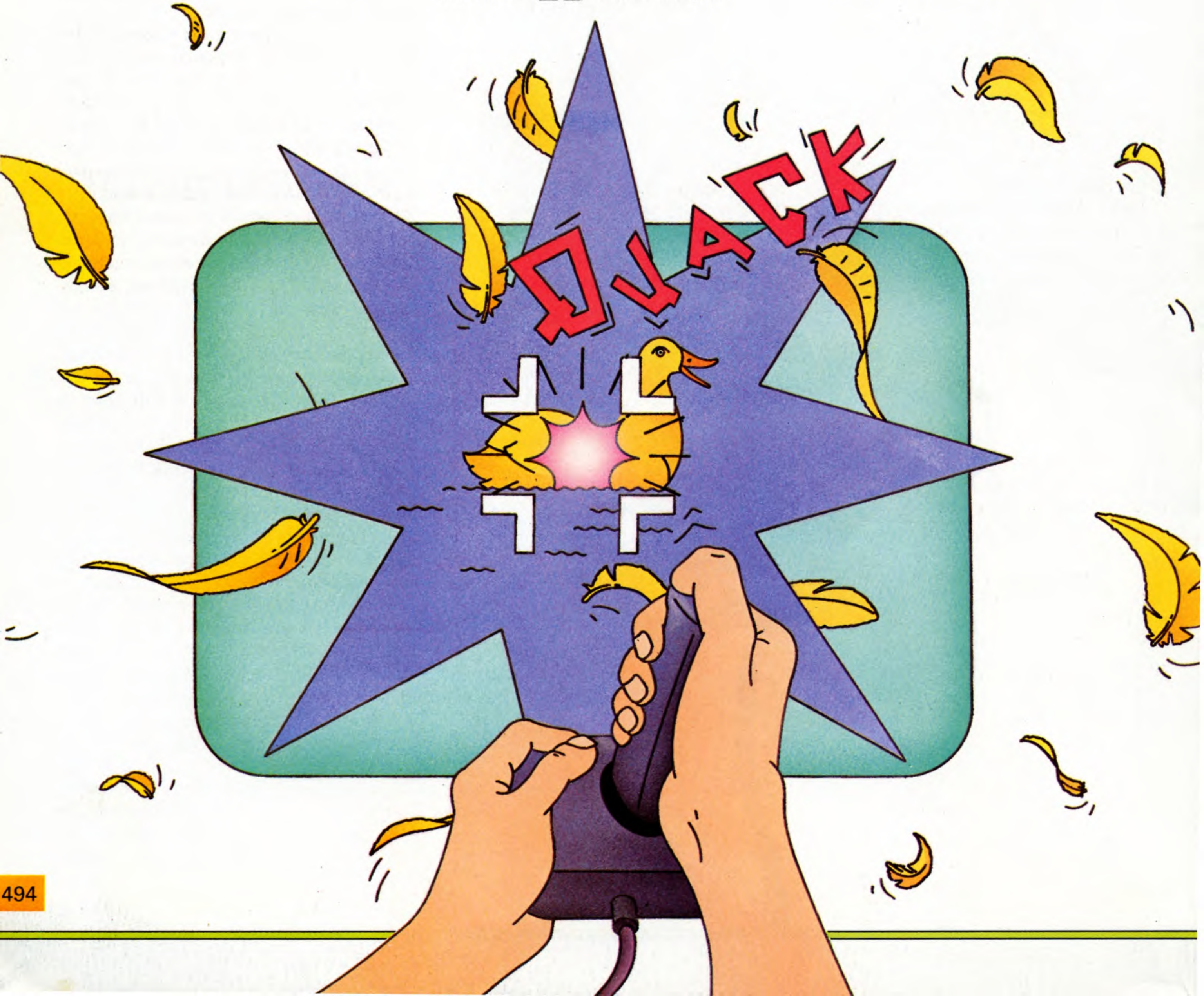
Should ATTR not return the duck's colour, the program continues to Line 23Ø, where a penalty of ten points is subtracted from the running points total. Next, Line 4ØØ checks if the time limit has elapsed, and jumps to Line 2ØØ if it hasn't. If more than four seconds have elapsed Line 41Ø blots out the duck.

After ten ducks have been shown on the screen, the program reaches Line 43Ø which compares the player's score with the high score. The high score is changed if the player's score is higher than the existing record. The high score and the current score are then displayed by Line 44Ø.

The program is completed by a very standard 'Do you want another go?' routine.

The Commodore 64 game is completed by adding these program lines:

```
10 GOSUB 330
50 D=(1064+INT(RND(1)*37)+1)
   +(INT(RND(1)*20)+1)*40:
   TI$="000000"
60 TU=TU+1: IF TU>10 THEN 260
70 PRINT "[CLR][DOWN]";SPC(30);
   "DUCK:";TU
80 POKE D,128:POKE D+1,129:
   POKE D+40,130:POKE D+41,131
90 PRINT "[CLR][π]";TAB(11);
   "TM:□";TI$
100 PRINT "[CLR][▒]SC:□□□□
   □□□□▌▌▌▌▌▌▌▌▌▌
   ▌▌▌▌";SC
190 IF ((JAND16)=0)=-1 THEN 240
200 IF TI$<"000006" THEN 80
210 S=10:ED=500:GOSUB 400
220 SC=SC-10
230 POKE D,32:POKE D+1,32:
   POKE D+40,32:POKE D+41,32:
   GOTO 50
240 IF Y<>D THEN SC=SC-10:
   S=30:ED=150:GOSUB 400:
   GOTO 80
250 SC=SC+(10-VAL(TI$))*5:
   FOR S=10 TO 30:ED=30:
   GOSUB 400:NEXT S:GOTO 50
260 IF SC>HS THEN HS=SC
270 PRINT "[♥][π]YOUR SCORE:";
   SC:PRINT"[▄]HIGH SCORE:";HS
280 PRINT TAB(15);"[■][■][◣]END OF
   GAME"
290 PRINT TAB(12);"ANOTHER GO? (Y/N)"
300 GET A$:IF A$="Y" THEN 20
310 IF A$<>"N" THEN 300
320 PRINT "[♥][◆]":POKE 53280,14:
   POKE 53281,6:POKE 53272,21:
   END
330 POKE 52,48:POKE 56,48
340 POKE 56334,0:POKE 1,35
350 FOR Z=0 TO 1023:POKE
   12288+Z,PEEK(53248+Z): NEXT Z
360 POKE 1,39:POKE 56334,1
370 FOR Z=0 TO 31:READ X:POKE
   13312+Z,X:NEXT Z:POKE
   53272,28:RETURN
380 DATA 14,27,63,31,15,7,15,31,0,0,
   0,0,0,192,112,188
390 DATA 31,29,30,15,3,1,1,3,206,30,
   124,248,224,64,64,224
400 V=54272:POKE V+24,15:
   POKE V+5,15:POKE V+6,248
410 POKE V+1,S:POKE V+4,17:
   FOR DE=1 TO ED:NEXT DE
420 POKE V+4,0:POKE V+5,0:
   POKE V+6,0:RETURN
```



Line 1Ø calls a subroutine located between Lines 33Ø and 37Ø which sets up the extra graphics needed for the game.

Line 33Ø clear some memory space for the graphics. Next the character ROM is copied into RAM, but first the interrupts must be switched off to allow you to do this without corruption (interrupts are explained in detail in a later article). Line 34Ø turns the interrupts off and switches in the character ROM ready for it to be copied in Line 35Ø. The character ROM is switched out, and the interrupts are switched on again by Line 36Ø.

The duck graphic is POKEd on to the screen by Line 37Ø from the DATA in Lines 38Ø and 39Ø. The subroutine then ends.

The program RETURNs to Line 5Ø which chooses at position for the duck at random. The timer is also set to zero. Next, the number of ducks, TU, is incremented, and is checked to make sure that no more than ten have been displayed. The number of ducks is displayed by Line 7Ø, and the duck graphic is POKEd on screen by Line 8Ø. The time and score are displayed by Lines 9Ø and 1ØØ.

Line 19Ø checks if the fire button has been pressed, and jumps to Line 24Ø if it has. The next check looks at the timer. If the time elapsed is less than six seconds, the program jumps back to Line 8Ø.

Line 21Ø sets up the variables for the sound effect subroutine at Lines 4ØØ to 42Ø. S and ED allow the same basic subroutine to be used to generate a number of different sounds.

The score and high score are displayed by Line 27Ø after the score has been calculated by Line 25Ø—the exact value of the score depends on how long the player takes to shoot the duck—and the high score is altered by Line 26Ø.

The section of program from Line 29Ø to Line 31Ø is a standard 'Another go?' routine of the type used in several previous games. If the player doesn't want another go, the screen colour, border colour and graphics mode are reset and the program ends at Line 32Ø.

The game is completed by adding these lines to your existing joystick routine:

```
1Ø DATA 14,27,63,31,15,7,15,31,Ø,Ø,Ø,
   Ø,Ø,192,112,188
2Ø DATA 31,29,3Ø,15,3,1,1,3,2Ø6,3Ø,
   124,248,224,64,64,224
3Ø FOR Z=Ø TO 31:READ X:POKE 7168
   +Z,X:NEXT Z
8Ø D=(77Ø2+INT(RND(1)*19)+1)
   +(INT(RND(1)*2Ø)+1)*22:
   TI$="ØØØØØØ"
9Ø TU=TU+1:IF TU>1Ø THEN 36Ø
1ØØ PRINT "■■■■■"TAB(7)
   "DUCK:"TU:FOR Z=1 TO 999:NEXT Z
11Ø PRINT "■■■■■"TAB(7)
   "□□□□□□□□□"
12Ø POKE D,Ø:POKE D+1,1:POKE D+22,2:
   POKE D+23,3
13Ø PRINT "■■■"TAB(11)"TM:□"
   TI$
14Ø PRINT "■■■SC:□□□□
   □□□□■■■■■■■■■■
   ■■■"SC
25Ø IF F=-1 THEN POKE 36877,13Ø:
   GOTO 3ØØ
26Ø IF TI$<"ØØØØØ5" THEN 12Ø
27Ø FOR Z=2ØØ TO 127 STEP-1:
   POKE 36875,Z:NEXT Z
28Ø SC=SC-1Ø
29Ø POKE D,32:POKE D+1,32:POKE
   D+22,32:POKE D+23,32:GOTO 8Ø
3ØØ IF Y<>D THEN SC=SC-1Ø:GOTO
   33Ø
31Ø SC=SC+(6-VAL(TI$))*1Ø:FOR
   S=2ØØ TO 127 STEP -1
32Ø POKE 36865,36+RND(1)*6:POKE
   36876,S:POKE 36865,38:NEXT S
33Ø POKE 36877,Ø
34Ø IF Y=D THEN 8Ø
35Ø GOTO 12Ø
36Ø IF SC>HS THEN HS=SC
37Ø PRINT "■■■YOUR SCORE:"SC:
   PRINT "■■HIGH SCORE:"HS
38Ø PRINT "■■■■■■■■■■
   END OF GAME"
39Ø PRINT "■■■■ANOTHER GO?
   (Y/N)"
4ØØ GET A$:IF A$="Y" THEN 5Ø
41Ø IF A$<>"N" THEN 4ØØ
42Ø PRINT "■■":POKE 36879,27:
   POKE 36869,24Ø
```

The Vic 20 program is very similar to the one written for the Commodore 64.

Lines 1Ø to 3Ø define the duck and display it on screen. Lines 8Ø to 14Ø are exactly the same as Lines 5Ø to 1ØØ in the 64 program, displaying the duck and the scores.

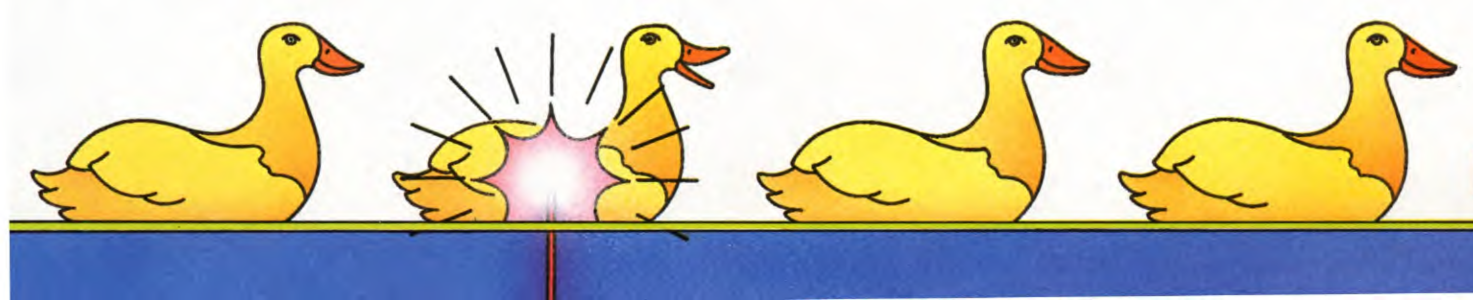
Line 25Ø makes a sound if the fire button is pressed and then jumps to Line 3ØØ. As the Vic 20 has a smaller screen grid than the 64, a shorter time limit of five seconds is set by Line 26Ø. Line 27Ø makes an 'out of time' sound if you're too slow before Line 28Ø subtracts ten from the score as a penalty for running out of time.

Line 29Ø rubs out the duck, and Line 3ØØ subtracts ten from the score if the player has missed. If the player has been successful Line 31Ø calculates the score—the additional points depend on how quickly the player has managed to pepper the duck full of holes. A 'successful kill' sound is made by the FOR ... NEXT loop in Lines 31Ø to 32Ø. Line 33Ø switches the sound off—the line also switches off the 'miss' sound. If there has been a hit, then Line 34Ø jumps back to Line 8Ø which replots the duck. If you missed, Line 35Ø sends the program back to line 12Ø instead which displays the next duck.

The score and high score are displayed by Line 31Ø after Line 36Ø has adjusted the high score if necessary. The remainder of the program is a standard 'Another go?' routine, followed by Line 42Ø resetting the screen colour and mode.

The following section of program completes the game of duckshoot for the BBC computer. Electron owners should use the next program instead. On both computers, delete the temporary Line 115.

```
1Ø HISC=Ø
4Ø F=Ø:SC=Ø:TI=Ø
5Ø X2=RND(1ØØØ)+14Ø:Y2=RND
   (8ØØ)+1ØØ
8Ø PRINT"SCORE□□□□□Ø"
9Ø VDU23,224,112,224,63,126,124,
   56,16,48
15Ø IF F=Ø THEN 17Ø
18Ø VDU4:PRINTTAB(1Ø,Ø);SC
   "□□□":VDU5
19Ø PROCDUCK
2ØØ GCOL3,3
21Ø IF F>1Ø THEN 42Ø
27Ø DEF PROCDUCK
28Ø GCOL3,2
29Ø IF F=Ø THEN 36Ø
3ØØ IF TIME>TI THEN 35Ø
```



```
310 IF V(1)>X-32 AND V(1)<X+32 AND
    V(2)>Y-32 AND V(2)<Y+32 AND
    (ADVAL(Ø) AND 1) THEN 34Ø
32Ø IF (ADVAL(Ø) AND 1) AND TIME>2Ø
    THEN SC=SC-2Ø:SOUND1,-15,1Ø,1
33Ø ENDPROC
34Ø SC=SC+3ØØ-TIME:
    SOUND1,-15,2ØØ,1
35Ø MOVEX,Y:VDU224
36Ø F=F+1
37Ø X=RND(1ØØØ)+1ØØ:Y=RND
    (8ØØ)+1ØØ
38Ø MOVEX,Y:VDU224
39Ø TIME=Ø
4ØØ TI=RND(2)*1ØØ+1ØØ
41Ø ENDPROC
42Ø MODE1
43Ø IF SC>HISC THEN HISC=SC
44Ø PRINTTAB(1Ø,1Ø)"HIGH SCORE□";
    HISC
45Ø PRINTTAB(1Ø,13)"SCORE
    □□□□□□";SC
46Ø INPUT''"PRESS RETURN"A
47Ø GOTO 3Ø
```

Lines 1Ø and 4Ø initialize variables for the high score, number of ducks that have been displayed, score and time elapsed.

A random screen position for the duck is chosen by Line 5Ø. The starting score is displayed by Line 8Ø, and the VDU in Line 9Ø sets up the duck graphic.

Line 15Ø is quite important because it prevents the program reaching Lines 16Ø and 17Ø and causing a nonsensical MOVE to be calculated—V and V2 both contain zeros at this stage. Once F has increased to one or more, V and V2 will contain suitable values, so the screen position in Lines 16Ø and 17Ø will now correspond to the joystick position.

Line 18Ø uses VDU4 to go back to the text cursor, to display the score. Finally, the VDU5 switches back to the graphics cursor ready for Line 19Ø to call PROCDUCK.

PROCDUCK is a large section of program stretching from Lines 27Ø to 41Ø dealing with printing and shooting the duck. First of all, Line 28Ø specifies yellow instead of white so that you will have some nice yellow ducks to shoot at. If this is the first duck, Line 29Ø makes the program jump to Line 36Ø.

The program continues to Line 3ØØ which checks if the time limit has expired. If the time limit hasn't expired, then Line 31Ø checks if the gunsight is pointing at the duck and the fire button on the joystick is being pressed. If you have written a program which uses both joysticks, you should AND ADVAL(Ø) with 2 instead of 1. Just in case the player presses the fire button at the moment the duck disappears Line 32Ø imposes a penalty of only 20 points for a miss if the shot is made more than 0.4 of a second late. If the player does miss, a 'missing sound' is made. The PROCedure ends at Line 33Ø.

If the duck was hit, Line 34Ø calculates the score according to the amount of time the duck has been on screen and a different sound is made.

The duck total is increased by 1 by Line 36Ø, and the duck's new position is chosen by Line 37Ø. The duck is put at its new position by Line 38Ø, and the timer is reset immediately afterwards by Line 39Ø. The duck may appear for either one or two seconds—chosen by Line 4ØØ—before the PROCedure ends at Line 41Ø.

The final section of program—Lines 42Ø to 47Ø—deals with the end of the game. After ten ducks have been displayed, Line 42Ø clears the screen and resets the colours. The score and high score are compared in Line 44Ø before Lines 45Ø and 46Ø display the high score and score that that game. Pressing RETURN will restart the program.

Users of the Electron will have to make these additions to the original joystick program:

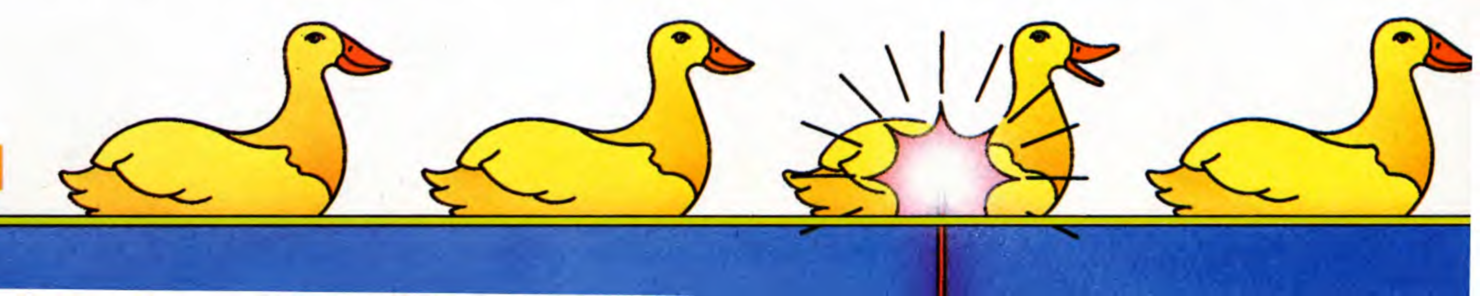
```
1Ø HISC=Ø
4Ø F=Ø:SC=Ø:TI=Ø
5Ø X2=RND(1ØØØ)+14Ø:Y2=RND
    (8ØØ)+1ØØ
8Ø PRINT"SCORE□□□□□Ø"
9Ø VDU23,224,112,224,63,126,124,
    56,16,48
15Ø IF F=Ø THEN 17Ø
18Ø VDU4:PRINTTAB(1Ø,Ø);SC
    "□□□":VDU5
19Ø PROCDUCK
2ØØ GCOL3,3
21Ø IF F>1Ø THEN 42Ø
27Ø DEF PROCDUCK
28Ø GCOL3,2
29Ø IF F=Ø THEN 36Ø
3ØØ IF TIME>TI THEN 35Ø
31Ø IF V(1)>X-32 AND V(1)<X+32 AND
    V(2)>Y-32 AND V(2)<Y+32 AND
    (?&FCCØ AND 16)=Ø THEN 34Ø
32Ø IF (?&FCCØ AND 16)=Ø AND TIME>2Ø
    THEN SC=SC-2Ø: SOUND1,
    -15,1Ø,1
33Ø ENDPROC
34Ø SC=SC+7ØØ-TIME:SOUND1,
    -15,2ØØ,1
35Ø MOVEX,Y:VDU224
36Ø F=F+1
37Ø X=RND(1ØØØ)+1ØØ:Y=RND
    (8ØØ)+1ØØ
38Ø MOVEX,Y:VDU224
39Ø TIME=Ø
4ØØ TI=RND(3)*1ØØ+4ØØ
41Ø ENDPROC
42Ø MODE1
43Ø IF SC>HISC THEN HISC=SC
44Ø PRINTTAB(1Ø,1Ø)"HIGH
    SCORE□";HISC
45Ø PRINTTAB(1Ø,13)"SCORE
    □□□□□□";SC
46Ø INPUT''"PRESS RETURN"A
47Ø GOTO3Ø
```

The program is extremely similar to the one designed to work with the BBC.

The differences lie in Lines 31Ø and 32Ø, which check for presses on the fire button in a different way, and Lines 33Ø and 34Ø, which make allowances for the slower processing speed of the Electron.

Adding the remainder of the program will give you the complete Duckshoot game:

```
8Ø FORK=1536TO2272 STEP32
9Ø READA,B:POKEK,A:POKEK+1,B
1ØØ NEXT
11Ø GET(Ø,Ø)-(13,11),D,G
12Ø GET (Ø,12)-(13,23),H,G
15Ø DX=RND(239)+1:DY=RND(178)+1
18Ø D=1Ø:SC=Ø
19Ø TIMER=Ø
21Ø PUT(DX,DY)-(DX+13,DY+11),
    D,OR
22Ø IF(PEEK(6528Ø)AND1)=Ø
    GOSUB2ØØØ:IFD<1 THEN 25Ø
23Ø IF TIMER<2ØØ THEN 2ØØ
24Ø D=D-1:IF D>Ø GOSUB3ØØØ:
    GOTO19Ø
25Ø CLS:PRINT@14Ø,"SCORE =";SC
```



```
26Ø IF SC>HI THEN HI=SC
27Ø PRINT@233,"HI-SCORE =";HI
28Ø PRINT@389,"ANOTHER GO ?(Y/N)"
29Ø A$=INKEY$:IF A$<>"Y" AND
   A$<>"N" THEN 29Ø
3ØØ IF A$="Y" THEN 13Ø
31Ø END
2ØØØ IFPPOINT(X,Y)=1 THEN 2Ø7Ø
2Ø1Ø PUT(X-6,Y-5)-(X+7,Y+5),
   H,PSET
2Ø2Ø PLAY"T5ØO4CEEEG"
2Ø3Ø GOSUB3ØØØ
2Ø4Ø SC=SC+25Ø-TIMER
2Ø5Ø D=D-1:TIMER=Ø
2Ø6Ø RETURN
2Ø7Ø PLAY"T25ØO1DDE"
2Ø8Ø SC=SC-1Ø
2Ø9Ø RETURN
3ØØØ PUT(DX,DY)-(DX+13,DY+11),
   B,PSET
3Ø1Ø PUT(X-8,Y-5)-(X+9,Y+5),
   S,PSET
3Ø2Ø DX=RND(239)+1:DY=RND
   (178)+1:RETURN
4Ø2Ø DATA 4,Ø,25,Ø,149,Ø,21,Ø,5,Ø,5,64,5,
   8Ø,21,148,22,148,22,84,21,84,5,8Ø
4Ø3Ø DATA 48,48,Ø,Ø,195,12,51,48,15,192,
   51,48,2Ø4,2Ø4,15,192,51,48,192,
   12,Ø,Ø,51,48
```

Two shapes are contained in DATA in Lines 4Ø2Ø and 4Ø3Ø—a duck, and a red splatter which will be used after the duck has been successfully disposed of. The DATA is READ and POKEd onscreen by Lines 8Ø to 1ØØ. Two more GETs are needed—in Line 11Ø the duck is stored in array D, and in Line 12Ø the splatter is stored in array H.

A random screen position for the duck is selected by Line 15Ø. The values of DX and DY are used by Line 21Ø to PUT the duck on the screen.

Line 18Ø sets the initial number of ducks to ten, and the score to zero. The timer is set to zero by Line 19Ø.

Presses on the fire button are detected by Line 22Ø. If you press the button on the right hand joystick, bit zero in memory location 6528Ø will be changed from 1 to Ø, whilst if you press the fire button on the left hand joystick bit one will be changed from 1 to Ø. Line 22Ø shows how you detect the changes in 6528Ø. PEEKing 6528Ø and ANDing—see page 288—the contents with 1 will give zero if the right hand fire button is depressed, whilst ANDing the contents with 2 will give zero if the left hand fire button is being pressed. Line 22Ø in its present form, then, checks for presses on the right hand fire button. If a press is detected, it jumps to the subroutine beginning at Line 2ØØØ which deals with hits and misses.

The final part of Line 22Ø checks if all ten ducks have been displayed and jumps to Line 25Ø if they have.

Once the fire button has been pressed, Line 2ØØØ looks at the colour of the pixel at the centre of the gunsight at that moment. If PPOINT—to be described later—finds that the colour is green, then the shot must have missed, and the program jumps to Line 2Ø7Ø which makes a miss sound. The penalty for missing is to lose ten points—they are subtracted by Line 2Ø8Ø. The subroutine ends at Line 2Ø9Ø.

If the colour at the centre of the gunsight isn't green, then the shot must have been on target. The program carries on to Line 2Ø1Ø which PUTs the splatter on the screen. To add to the effect, a sound is PLAYed by Line 2Ø2Ø. Next, the subroutine at Line 3ØØØ is called. Its function is to blank out the splatter and replace it with the gunsight ready for the game to continue. Line 3Ø2Ø picks a new random position for the duck.

Line 2Ø4Ø calculates the score. The additional score depends on how quickly the player has succeeded in shooting the bird, so values between 50 and 250 may be added—the game stops when the TIMER value exceeds 2ØØ. Having calculated the score, the number of ducks is decremented, and the timer is reset to zero. The subroutine ends at Line 2Ø6Ø this time.

After completing the subroutine, the program RETURNs to Line 23Ø, where the time elapsed is checked. If the TIMER reading is below 200 the program jumps back to Line 2ØØ. If the time limit has elapsed, on the other hand, Line 24Ø decrements the number of ducks, and checks if any remain. If they do, the duck is blanked out by the subroutine starting at Line 3ØØØ. Line 3Ø2Ø selects the position of the next duck.

If all the ducks have been used up the scores and high scores are shown. Line 25Ø automatically switches back to the text screen and displays the player's score. The score is checked against the high score, and altered if necessary by Line 26Ø, before Line 27Ø displays the high score.

Finally, Lines 28Ø to 31Ø are a familiar 'Another go?' routine.

Q+A

What do I have to do to change the duck graphic into a different target?

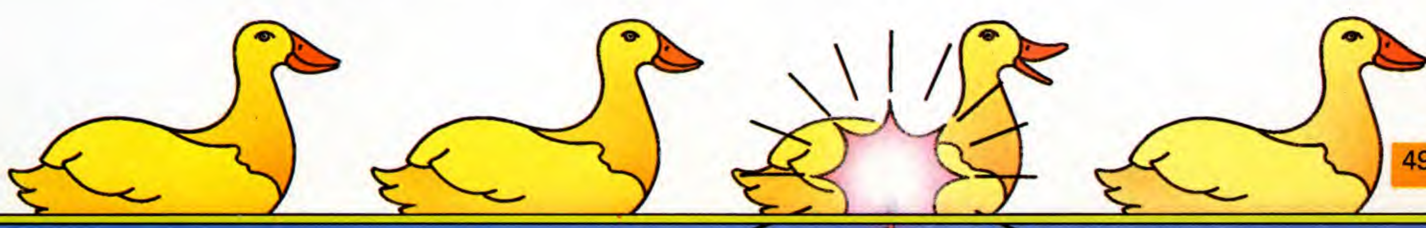
The answer depends on which machine you have. Basically, it is a matter of altering the DATA, and changing the lines that PRINT the UDG on the screen.

The Spectrum program uses a block of four UDGs defined in Lines 1ØØØ to 1Ø3Ø. If you use a larger UDG, Line 18Ø, which PRINTs them on the screen, will also need changing.

Commodore users will have to change the DATA. If your new UDG calls for more DATA, the FORs in Line 37Ø (for the 64), and Line 3Ø (for the Vic), will have to be changed too.

The duck graphic in the Acorn programs occupies an eight by eight pixel grid, and you may well find that a graphic of this size is too small. If you need a larger character, look at pages 42 to 43 where you will find instructions on how to define and display larger UDGs. If you want to use a 16 × 16 pixel grid, for example, you would set UDG$=CHR$224+CHR$225+CHR$1Ø+CHR$8+CHR$8+CHR$226+CHR$227, having defined CHR$224 to 227 using VDU23. You should then substitute PRINT UDG$ for VDU224 in Lines 35Ø and 37Ø. The hit check in Line 31Ø will also have to be changed, change the number 32 to 64.

If you have a Dragon or Tandy it's quite simple to create any size of UDG you wish. Change the DATA and the FOR ... NEXT loop which READs it, and the coordinates of the GETs and PUTs in the program.



EXTEND YOUR TYPING

Move on from learning the keyboard and put your typing skills to a real test by copying longer phrases generated on the screen by the computer

If you practised the earlier stages of the typing course, you should by now be familiar with all the keyboard characters, and be able to type words and figures accurately and at a steady pace. Up to this point, the course is equally beneficial to those who wish to key in programs, as well as those who are more interested in writing letters or in word processing.

But now you have the opportunity to practise your skills on an extended piece of text. You will still gain in fluency on the keyboard, but you will also get more experience at using the computer as a writing tool.

It's very important at this stage that you maintain the same touch typing technique that you have practised throughout the earlier parts of the course. Try to type by feel alone, without looking at the keyboard—using the home keys to give you a reference point all the time. And try to maintain a steady rhythm, rather than a high speed that is erratic. The golden rule is to start off at a speed which you can manage at a steady pace, then build this up gradually.

HOW TO USE THE PROGRAM

When you RUN the program a menu appears on the screen giving you the option of two tests. The first test displays sentences, randomly generated from DATA statements in the program. The second test requires rather more work from you. It allows you to practise on longer passages but you have to enter the passage first. Once entered, the computer displays it on the screen as before. In both tests you have to type the sentence or passage as it appears on the screen, and afterwards the program displays your typing speed in words per minute and tells you the number of errors you made.

The program also allows you to choose what happens when you make an error—that is, whether you want to be able to use the BACKSPACE or DELETE key to correct your work or not.

If you choose to use the BACKSPACE or DELETE option, then, except on the Acorns, any errors have to be corrected as soon as they are made.

If you choose *not* to use the BACKSPACE or DELETE then, on all but the Acorns, the computer waits for you to enter the correct character before you can continue.

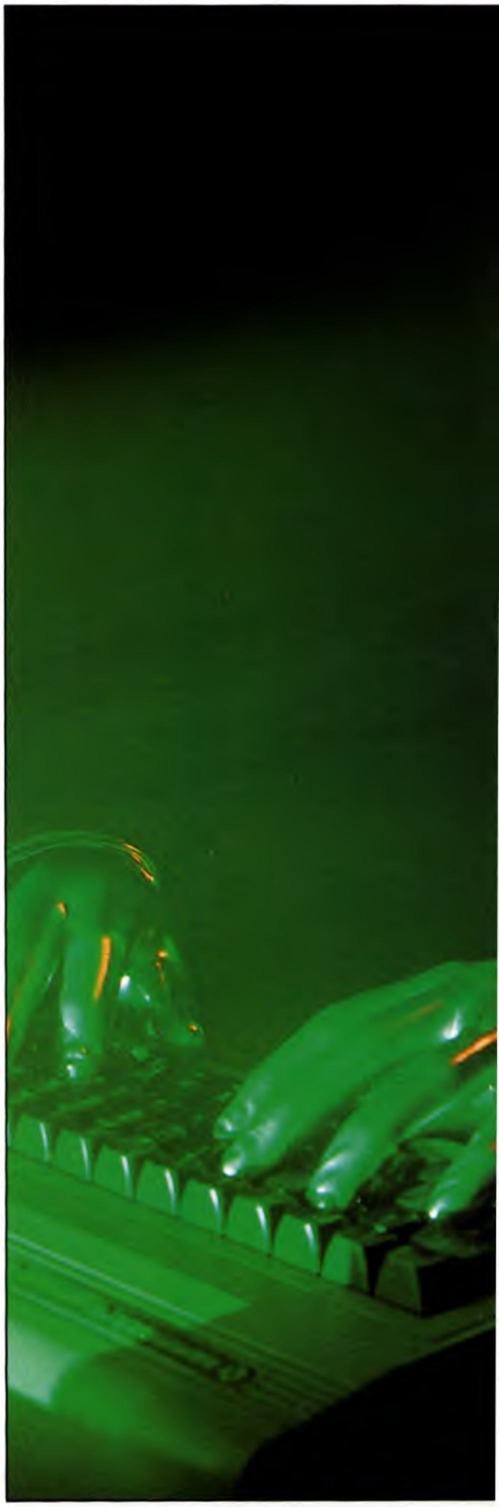
The Acorn program works slightly differently. In the first option you do not *have* to delete errors, so if you are concentrating on improving speed and typing rhythm they can be left as they are. In the second option the cursor continues to move on, so there's no way of correcting the errors. This is more useful if you want to concentrate on improving your accuracy.

Then, again on the Acorns, when your sentence is printed out at the end, the errors are highlighted in a different colour so you can see which keys you need more practice on.

After an hour or so at the keyboard, you will find that the sentences have become monotonous—there are only a few variations. If you like you can change the phrases in the program for others of your own. To do this, break into the program and LIST the section containing the DATA statements. These are in Lines 15ØØ onwards on the Spectrum, Dragon and Tandy, Lines 1ØØØ onwards for the Commodores and Lines 77Ø onwards for the Acorns.

Overwrite the phrases with your own, but remember to start and end phrases containing commas with double quotes("). If you do not, the computer will divide the phrase at the comma and treat each part as a separate phrase. You should also keep the same number of phrases in each section. If you do not feel sufficiently confident to make up your own phrases so that when combined they make some kind of sense, then you can vary the exercise by opting to key in whole passages of text.

A passage can be up to 255 characters long on the Spectrum, Dragon and Tandy, about 245 on the Acorns, 80 characters on the Vic 20 and 40 on the Commodore 64. You can enter three passages, except on the Acorn micros, where you can enter about 85. To enter the passages (except on the Acorn), select option 2 and respond to the prompts on the screen. Passages with commas or colons (:) should start and end with double quotes ("). Once you have entered three passages into the DATA statements, you can select any one to



- LEARNING TO TYPE REAL SENTENCES
- HOW THE PROGRAM WORKS
- USING THE DELETE OR BACK-SPACE KEY TO CORRECT ERRORS
- USING THE PROGRAM TO PRACTISE LONGER PIECES OF TEXT
- RULES FOR CORRECT PRESENTATION



practise on by choosing option 2.

To enter passages on Acorn micros, ESCAPE from the program and LIST. Then enter each passage as a DATA statement, starting at Line 84Ø. You could enter five or even ten passages to begin with, so that when you RUN the program and select option 2, one of these will be selected at random for you to type.

The only other option is whether or not to have sound, which is available in all except the Spectrum programs. Acorn users can choose to turn off the sound when they RUN the program, but others can merely turn down the volume control on the TV set.

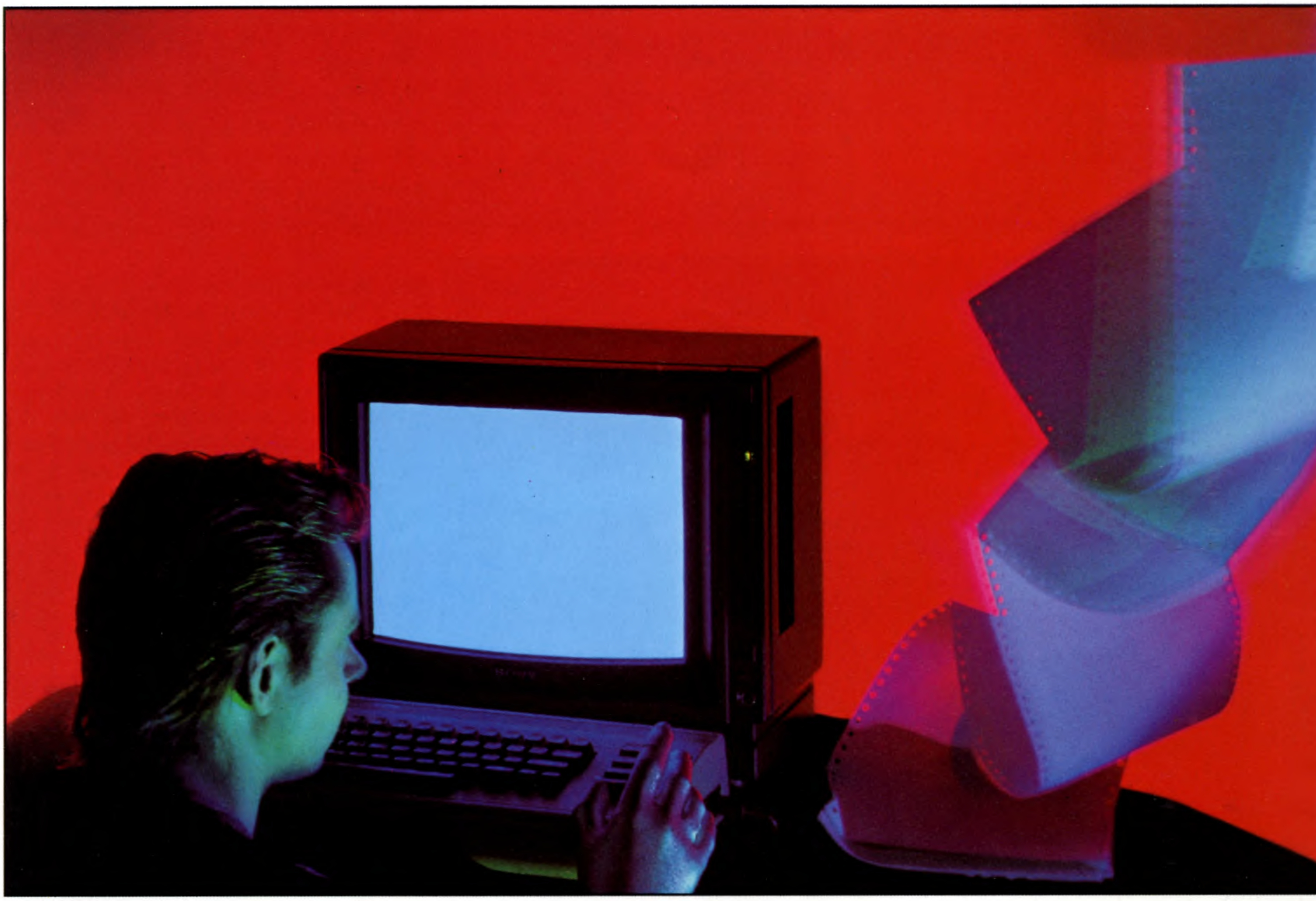
Q+A

Should I follow any rules when I make up new data statements for the program, or will any sentences do?

This depends on how well you can type at present. Provided you have worked through each stage of the typing course you should be able to type equally well with all your fingers—including the little ones. However, the little fingers do tend to need more practice than the others so it's as well to make sure that the words in the DATA statements include plenty of Qs, As, Zs, Ps and Ls. By making up different sets of words you can give extra practice to any of your fingers.

You should mix in these words and phrases with others that cover the whole of the keyboard. The old favourite 'A quick brown fox jumps over the lazy dog' is always useful, so is 'Pack my box with five dozen liquor jugs'. You'll no doubt be able to invent similar sentences of your own.

At present, the program combines three phrases to form a more or less meaningful sentence. If you find this too difficult you could enter short sentences so the program combines them into paragraphs.



```
10 POKE 23561,Ø
20 CLS
25 DIM t$(3,255): DIM t(3): LET df=Ø
30 PRINT AT 7,7;"WHICH TEST (1 OR 2)?"
40 PRINT AT 1Ø,9;"TYPE 'Ø' TO QUIT"
50 LET a$=INKEY$: IF a$<"Ø" OR
   a$>"2" THEN GOTO 5Ø
60 IF a$="Ø" THEN STOP
70 GOSUB VAL a$*1ØØØ
80 CLS : PRINT AT 15,4;"Words per
   minute=□";LEN c$*(INT ((5ØØ/(PEEK
   23672+256*PEEK 23673))*1ØØ)/1ØØ)
90 PRINT AT 17,6;"Number of errors=□";e
1ØØ GOTO 3Ø
1ØØØ CLS : PRINT "□Do you want to be able
   to use□□□the DELETE key (y/n)?"
1Ø1Ø LET a$=INKEY$: IF a$<>"n" AND
   a$<>"y" THEN GOTO 1Ø1Ø
1Ø2Ø LET e=Ø: LET d=Ø: IF a$="y" THEN
   LET d=1
1Ø3Ø CLS
1Ø4Ø LET c$="": RESTORE : FOR k=1 TO
   4: LET r=INT (RND*3)+1: FOR j=1 TO 3
1Ø5Ø READ b$: IF j=r THEN LET
   c$=c$+b$
1Ø6Ø NEXT j: NEXT k
1Ø7Ø PRINT c$: PRINT : PRINT
1Ø8Ø LET pp=Ø
1Ø9Ø LET a$=INKEY$: IF a$="" THEN
   GOTO 1Ø9Ø
11ØØ POKE 23672,Ø: POKE 23673,Ø: PAUSE
   Ø: GOTO 112Ø
111Ø PAUSE Ø
1115 LET a$=INKEY$: IF a$="" THEN
   GOTO 111Ø
112Ø IF a$<>c$(pp+1) AND d=Ø THEN
   GOTO 117Ø
113Ø PRINT a$;CHR$ 95;CHR$ 8;: LET
   pp=pp+1
114Ø IF a$<>c$(pp) THEN GOTO 117Ø
115Ø BEEP .Ø1,3Ø: IF pp=LEN c$ THEN
   RETURN
116Ø GOTO 111Ø
117Ø BEEP .Ø5,-1Ø: LET e=e+1
118Ø IF d=Ø THEN GOTO 111Ø
119Ø PAUSE Ø: LET a$=INKEY$: IF
   a$<>CHR$ 12 THEN GOTO 119Ø
12ØØ LET pp=pp-1: PRINT CHR$ 8;CHR$
   95;"□";CHR$ 8;CHR$ 8;: GOTO 111Ø
121Ø GOTO 113Ø
15ØØ DATA "The mangy dog that walks on three
   legs□","It is a fact that anyone□","When
   the time is right, the elephant□"
151Ø DATA "might browse under□","will be
   able to sit on□","can jump upon□"
152Ø DATA "the wobbly box with a hole on
   top□","the leaning tower of Pisa□","any
   one of the farm buildings□"
153Ø DATA "and bring it crashing to the
   floor.","without fear of a big surprise.","until
   closing time at the zoo."
2ØØØ CLS : IF df=1 THEN GOTO 2Ø15
2ØØ5 LET df=1
2Ø1Ø FOR n=1 TO 3: INPUT "Input passage
   number□";(n)' LINE r$: LET t(n)=LEN r$:
   LET t$(n)=r$: NEXT n
2Ø15 INPUT "Which passage would you like to
   type in (1 to 3)?□"'p
2Ø17 IF p>3 OR p<1 THEN GOTO 2Ø15
2Ø18 LET c$=t$(p, TO t(p))
2Ø2Ø CLS : PRINT "Do you wish to be able to
   use□□□the DELETE key (y/n)?"
2Ø3Ø LET a$=INKEY$: IF a$<>"y" AND
   a$<>"n" THEN GOTO 2Ø3Ø
2Ø4Ø LET d=Ø: LET e=Ø: IF a$="y" THEN
   LET d=1
2Ø5Ø CLS : GOSUB 1Ø7Ø: RETURN
```



When you key this program, be careful not to confuse some of the symbols at the start of some of the PRINT statements and DATA statements. For example, at Line 10, the large 'o' before 'HICH TEST' is obtained by pressing SHIFT and 'W'—not letter 'o'; similarly, the symbol in Line 20 is a SHIFTed T:

```
5 PRINT "♥"CHR$(14)CHR$(8)
10 PRINT "▣◯HICH TEST (1 or 2)?"
20 PRINT "┳YPE 'Ø' TO QUIT"
30 GET A$:IF A$<"Ø" OR A$>"2" THEN
   3Ø
40 IF A$="Ø" THEN PRINT
   "♥"CHR$(142)CHR$(9):END
50 ON VAL(A$) GOSUB 1ØØ,2ØØØ
60 PRINT "♥"
70 PRINT "◯ORDS PER MINUTE":
   PRINT "=";INT((LEN(C$)/6)/
   TI*6ØØØ)
80 PRINT "▣◪UMBER OF ERRORS":
   PRINT "=";E
90 GOTO 1Ø
100 PRINT "♥⊟O YOU WANT TO BE ABLE
   TO USE THE DELETE KEY (Y/N)?"
110 GET A$:IF A$<>"Y" AND A$<>"N"
   THEN 11Ø
120 E=Ø:D=Ø:IF A$="Y" THEN D=1
130 PRINT "♥"
140 C$="":RESTORE:FOR K=1 TO
   4:R=INT(RND(1)*3)+1:FOR J=1 TO 3
150 READ B$:IF J=R THEN C$=C$+B$
160 NEXT J,K
170 PRINT C$;"▣▣▣"
180 PP=Ø:PRINT"▣";
190 GOSUB 31Ø
200 TI$="ØØØØØØ":GOTO 22Ø
210 GOSUB 31Ø
220 IF A$<>MID$(C$,PP+1,1) AND D=Ø
   THEN 27Ø
230 PRINT A$;:PP=PP+1
240 IF A$<>MID$(C$,PP,1) THEN 27Ø
250 SS=25Ø:GOSUB 3ØØØ:IF PP=LEN(C$)
   THEN RETURN
260 GOTO 21Ø
270 SS=13Ø:GOSUB 3ØØØ:E=E+1
280 IF D=Ø THEN 21Ø
290 GET A$:IF A$<>CHR$(2Ø) THEN 29Ø
300 PP=PP-1:PRINT A$;:GOTO 21Ø
310 POKE 198,Ø
320 GET A$:IF A$="" THEN 32Ø
330 IF A$<CHR$(32) OR (A$>CHR$(127)
   AND A$<CHR$(161)) THEN 32Ø
340 RETURN
1000 DATA "┳HE MANGY DOG THAT
   WALKS ON THREE LEGS□"
1010 DATA "◹T IS A FACT THAT
   ANYONE□","◯HEN THE TIME IS RIGHT,
   THE ELEPHANT□"
1020 DATA "MIGHT BROWSE UNDER□",
   "WILL BE ABLE TO SIT ON□","CAN
   JUMP UPON□"
1030 DATA "THE WOBBLY BOX WITH A
   HOLE ON TOP□","THE LEANING TOWER
   OF PISA□"
1040 DATA "ANY ONE OF THE FARM
   BUILDINGS□"
1050 DATA "AND BRING IT CRASHING TO
   THE FLOOR."
1060 DATA "WITHOUT FEAR OF A BIG
   SURPRISE.","UNTIL CLOSING TIME AT
   THE ZOO."
2000 IF C$(1)<>"" OR C$(2)<>"" OR
   C$(3)<>"" THEN 2015
2005 PRINT "♥◹NPUT THE PASSAGES
   YOU WISH TO TYPE.":
   FOR Z=1 TO 3:PRINT
   "▣▟ASSAGE:";Z
2010 INPUT C$(Z):NEXT Z
2015 PRINT "♥⊟O YOU WISH TO BE ABLE
   TO USE THE DELETE KEY (Y/N)?"
2020 GET A$:IF A$<>"Y" AND
   A$<>"N" THEN 2020
2025 D=Ø:E=Ø:IF A$="Y" THEN D=1
2030 PRINT "♥⊟NTER PASSAGE.
   (1-3)?"
2040 GET A$:C$=C$(VAL(A$)):
   IF A$<"1" OR A$>"3" THEN 2Ø3Ø
2050 PRINT "♥":GOSUB 17Ø:
   RETURN
3000 POKE 54296,15:POKE 54277,9:
   POKE 54273,SS
3010 POKE 54276,33:FOR DD=1 TO 4Ø:NEXT
   DD:POKE 54276,Ø:POKE 54277,Ø:RETURN
```

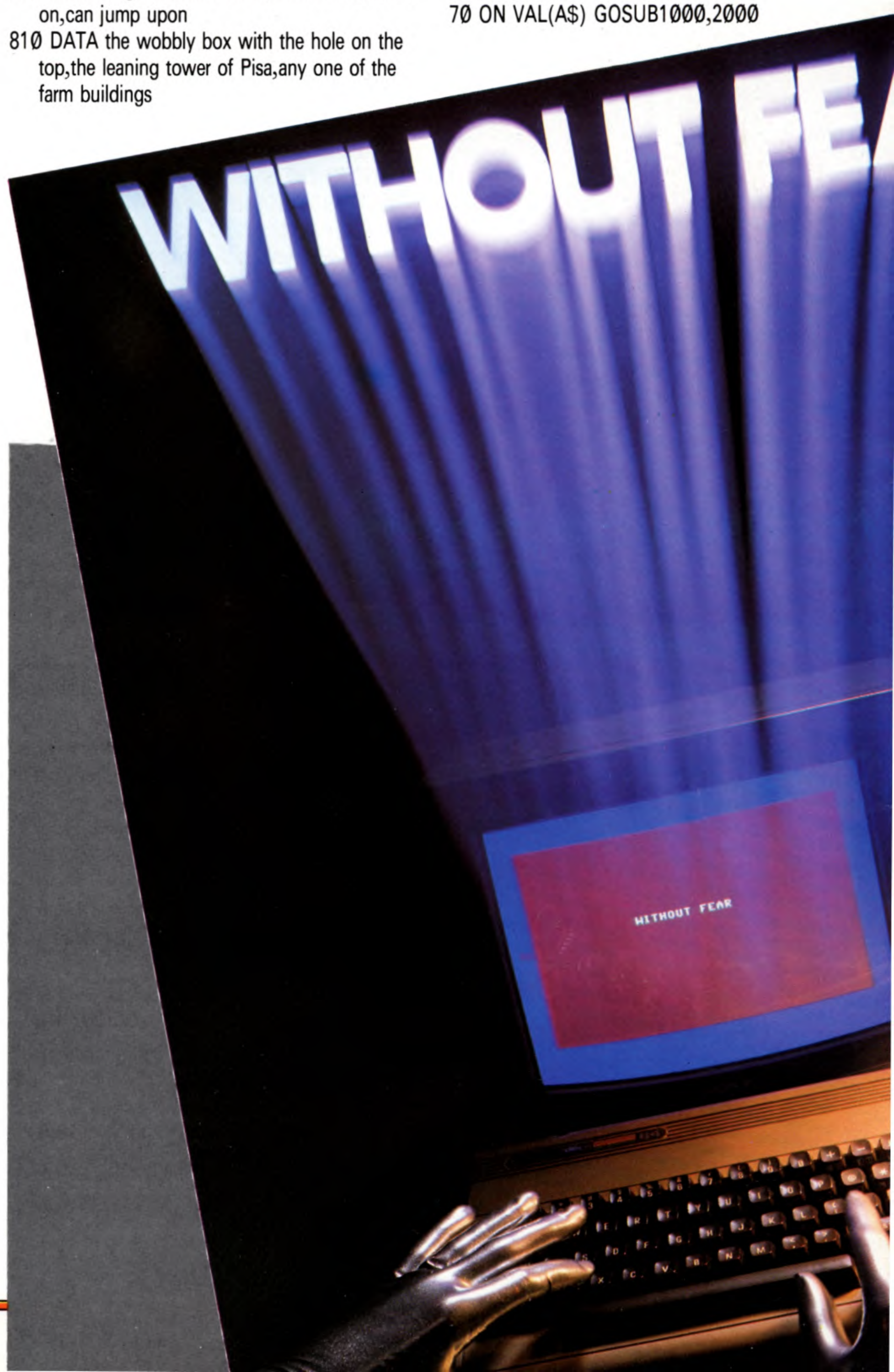
The program is the same as for the Commodore 64, except that the last two lines should be as follows:

```
3000 POKE 36878,15:POKE 36876,SS:FOR
   DD=1 TO 1Ø:NEXT DD
3010 POKE 36876,Ø:RETURN
```

```
10 SUB=3
20 NUM=SUB*4
30 ON ERROR GOTO 74Ø
40 MODE 1
50 VDU 23,224,255,255,255,255,
   255,255,255,Ø
60 GCOL3,3
70 DIM A$(1ØØ):FOR T=1 TO 1ØØ:
   READ A$(T):NEXT
80 N=T-NUM-1:D=1:S=-15
90 CLS:PRINTTAB(1Ø,9)"MAIN MENU"
100 PRINTTAB(1Ø,12)"1) SENTENCES"
110 PRINTTAB(1Ø,14)"2) PASSAGE"
120 PRINTTAB(1Ø,16)"3) SOUND + DELETE
   OPTIONS"
130 G=GET-48:IF G<1 OR G>3 THEN
   13Ø
140 IF G=1 THEN PROCSELECT:
   PROCINPUT1:PROCDISPERR:GOTO 9Ø
150 IF G=2 AND N>Ø THEN PROCPASS:
   PROCINPUT1:PROCDISPERR:GOTO 9Ø
160 IF G=3 THEN PROCERRHANDLE
170 GOTO 9Ø
180 DEF PROCINPUT1
190 CE=Ø:B2$=""
200 CLS:COLOUR2:PRINT''A$TAB
   (Ø,12):COLOUR3
210 VDU5:MOVE POS*32,1Ø23-(VPOS-
   1Ø)*32:VDU224,4
220 TIME=Ø
230 *FX15,1
240 FOR T=1 TO LEN(A$)
250 B$=GET$
260 IF NOT(D=1 AND B$=CHR$(127))
   THEN 29Ø
270 IF T>1 THEN VDU5:MOVE POS*32,
   1Ø23-(VPOS-1Ø)*32:VDU224,4:
   VDU ASC(B$):VDU5:MOVE POS*32,
   1Ø23-(VPOS-1Ø)*32:VDU224,4:
   CE=CE+1:T=T-1:B2$=LEFT$
   (B2$,T-1)
280 GOTO 25Ø
```

```
290 IF ASC(B$)<32 OR ASC(B$)>126
    THEN 25Ø ELSE VDU5:MOVE
    POS*32,1Ø23-(VPOS-1Ø)*32:
    VDU224,4:PRINTB$;:VDU5:
    MOVE POS*32,1Ø23-(VPOS-1Ø)
    *32:VDU224,4
3ØØ B2$=B2$+B$
31Ø IF B$=MID$(A$,T,1) THEN 33Ø
32Ø SOUND1,S,2Ø,1
33Ø NEXT
34Ø FINTIME=INT(TIME/1ØØ)
35Ø ENDPROC
36Ø DEF PROCDISPERR
37Ø VDU 14
38Ø CLS:E=Ø:PRINT'
39Ø FOR T=1 TO LEN(A$)
4ØØ D$=MID$(B2$,T,1)
4Ø5 COLOUR3
41Ø IF MID$(A$,T,1)<>D$ THEN
    COLOUR1:E=E+1:IF D$="□" THEN
    D$=CHR$(224)
42Ø PRINTD$;
43Ø NEXT
44Ø COLOUR3
45Ø PRINT''''"ACCURACY :"''TAB(1Ø)
    "YOU LEFT□";E;"□ERROR(S)"
46Ø IF D=1 THEN PRINTTAB(1Ø)
    "AND YOU CORRECTED□";CE;
    "□ERROR(S)"
47Ø PRINT'''"SPEED :"''TAB(1Ø);
    LEN(A$);"□LETTERS IN□";
    FINTIME;"□SECONDS"
48Ø PRINT'"THAT'S□";INT(LEN(A$)/
    6/FINTIME*6Ø);"□WORDS
    PER MINUTE"
49Ø VDU 15
5ØØ *FX15,1
51Ø INPUT''"PRESS RETURN FOR MENU",A$
52Ø ENDPROC
53Ø DEF PROCERRHANDLE
54Ø CLS
55Ø PRINT''''"DO YOU WANT SOUND (Y/N)"
56Ø G=GET AND &5F:IF G=89 THEN
    S=-15:GOTO 59Ø
57Ø IF G<>78 THEN 56Ø
58Ø S=Ø
59Ø PRINT''"DO YOU WANT TO USE DELETE
    KEY (Y/N)"
6ØØ G=GET AND &5F:IF G=89 THEN
    D=1:GOTO 63Ø
61Ø IF G<>78 THEN 6ØØ
62Ø D=Ø
63Ø ENDPROC
64Ø DEF PROCSELECT
65Ø A$=A$(RND(SUB))
66Ø FOR T=2 TO 4
67Ø A$=A$+"□"+A$(RND(SUB)+
    (T-1)*SUB)
68Ø NEXT
69Ø ENDPROC
7ØØ DEF PROCPASS
71Ø IF N=1 THEN A$=A$(NUM+1):
    ENDPROC
72Ø A$=A$(RND(N)+NUM+1)
73Ø ENDPROC
74Ø IF ERR<>42 THEN REPORT:
    PRINT"□AT LINE□";ERL
742 *FX2Ø2,32,Ø
745 IF ERR<>42 THEN END
75Ø GOTO 8Ø
77Ø DATA The mangy dog that walks on three
    legs,It is a fact that anyone,"When the time is
    right, the elephant"
79Ø DATA might browse under,will be able to sit
    on,can jump upon
81Ø DATA the wobbly box with the hole on the
    top,the leaning tower of Pisa,any one of the
    farm buildings
83Ø DATA and bring it crashing to the
    floor.,without fear of a big surprise.,until
    closing time at the zoo.
```

```
1Ø CLEAR2ØØØ
2Ø CLS:DIMA$(2)
3Ø PRINT@1Ø1,"WHICH TEST
   (1 OR 2) ?"
4Ø PRINT@168,"TYPE (Ø) TO QUIT"
5Ø A$=INKEY$:IF A$<"Ø"ORA$>"2"
   THEN5Ø
6Ø IFA$="Ø"THENCLS:END
7Ø ON VAL(A$) GOSUB1ØØØ,2ØØØ
```



```
80 CLS:PRINT@448,USING"WORDS PER
   MINUTE= ###.##";
   LEN(C$)*500/TIMER
90 PRINT@480,"NUMBER OF ERRORS
   =";E;
100 POKE329,255:GOTO 30
1000 CLS:PRINT"□DO YOU WISH TO BE
   ABLE TO USE□ □ □THE BACKSPACE KEY
   (Y/N) ?"
1010 A$=INKEY$:IF A$<>"N"
   ANDA$<>"Y" THEN1010
1020 E=0:D=0:IF A$="Y" THEN D=1
1030 CLS:POKE329,0
1040 C$="":RESTORE:FORK=1 TO
   4:R=RND(3):FORJ=1TO3
1050 READB$:IF J=R THEN C$=C$+B$
1060 NEXTJ,K
1070 PRINTC$
1080 PP=0
1090 A$=INKEY$:IF A$="" THEN1090
1100 TIMER=0:GOTO 1130
1110 A$=INKEY$:IF A$="" THEN 1110
1120 IF A$<>MID$(C$,PP+1,1)
   ANDD=0 THEN 1170
1130 PRINT@PP+256,A$:PP=PP+1
1140 IF A$<>MID$(C$,PP,1) THEN 1170
1150 SOUND200,1:IF PP=LEN(C$) THEN
   RETURN
1160 GOTO 1110
1170 SCREEN0,1:SOUND10,1:E=E+1
1180 IF D=0 THEN 1110
1190 A$=INKEY$:IF A$<>CHR$(8)
   THEN 1190
1200 PP=PP-1:PRINT@256,MID$
   (C$,1,PP):GOTO1110
1210 GOTO 1130
1500 DATA The mangy dog that walks on three
   legs□,It is a fact that anyone□,"When the
   time is right, the elephant□"
1510 DATAmight browse under□,will be able to
   sit on□,can jump upon□
1520 DATAthe wobbly box with the hole on
   top□,the leaning tower of Pisa□,any one of
   the farm buildings□
1530 DATAand bring it crashing to the
   floor.,without fear of a big surprise.,until
   closing time at the zoo.
2000 CLS:P=0:IF A$(0)=""ANDA$
   (1)=""ANDA$(2)="" THEN 2090
2010 PRINT"□DO YOU WISH TO USE A
   PREVIOUSLY ENTERED PASSAGE (Y/N) ?"
2020 A$=INKEY$:IF A$<>"Y" AND
   A$<>"N" THEN 2020
2030 IF A$="Y" THEN 2120
2040 IFA$(P)="" THEN 2090
2050 P=P+1:IFP<3 THEN 2040
2060 PRINT"□ALL THREE PASSAGES KEYED.
   WHICH□ □ □ONE DO YOU WISH TO
   OVERWRITE□ □ □ □(1-3) ?□";
2070 A$=INKEY$:IF A$<"1"
   ORA$>"3" THEN2070
2080 P=VAL(A$)-1:PRINTA$:PRINT
2090 POKE329,0:PRINT"□INPUT THE
   PASSAGE YOU WISH TO□ □ □TYPE -"
2100 LINEINPUT A$(P)
2110 GOTO 2150
2120 CLS:PRINT" WHICH PASSAGE DO YOU
   WISH TO□ □ □ □USE□ □
   (1-3) ?"
2130 A$=INKEY$:IFA$<"1"
   ORA$>"3" THEN 2130
2140 P=VAL(A$)-1:IFA$(P)=""
   THEN2120
2150 CLS
2160 POKE329,255:CLS:PRINT"□DO YOU
   WISH TO BE ABLE TO USE□ □ □THE
   BACKSPACE KEY (Y/N) ?"
2170 A$=INKEY$:IFA$<>"Y"AND
   A$<>"N" THEN 2170
2180 D=0:E=0:IF A$="Y" THEN D=1
2190 CLS:POKE329,0:C$=A$(P):
   GOSUB1070:RETURN
```



T

The program is the same as for the Dragon, except that the first number of the POKEs at Lines 100, 1030, 2090, 2160 and 2190 should be 282, instead of 329.

Microtip

Presentation

Given the ease with which you can correct errors on a computer keyboard, you should be able to print clean copies, without smudges from correcting fluid or feint characters caused by erasing errors with correcting paper. The final appearance of your typing, however, depends on the care you take in laying it out on the paper.

A well laid out document has margins of at least 3 cm on both sides of the paper, and the body of type should lie centrally from top to bottom. If you use a word processing program, this will have provision to set the margins and number of lines per page, but on your own programs, and the one listed here, you can set these limits by PRINT TAB or PRINT AT statements.

Just as important as margins is the use of line spaces to break up the block of type into paragraphs. This helps to make the text easy to read and understand. Paragraphs can be indented—with the first letter of the first line starting about two spaces from the margin—but modern practice is to type in block. In both methods, the paragraphs are separated by at least one line space (a blank line) but block paragraphs are not indented. If you have only a short passage or letter to type, start well down the page, have single spaces between each line and, say, treble line spacing between paragraphs. After a little practice, you should be able to judge where to start so that the passage or letter is centred on the page.

CHOOSING STORAGE METHODS

Data storage for home computers usually means a choice between cheap but well supported tape systems and expensive but powerful disk systems. Each has its advantages

All home computers use specific areas of what is called random access memory to store program information that's either keyed in or loaded from a *data storage device* of one kind or another. Storage devices are necessary because usable areas of memory are 'volatile'—information here simply disappears when the computer is switched off.

Several types of *tape* and *disk* storage device are available for use with home computers. The more expensive of these are usually both versatile and powerful enough to provide computing facilities which really do give home computers impressive capabilities—making them very much more suitable for educational and business uses.

These systems are all capable of providing the essential *permanent* storage without which computers couldn't function in the way we now accept as normal.

TAPE STORAGE

The favourite method of program storage for home computers is the use of ordinary tape cassette recorders as this represents a workable compromise between operational convenience and low cost. And, as far as the average user is concerned, this is all that really matters.

Low cost audio cassette recorders are often quite adequate for the job and there is a virtually limitless selection of software available in cassette form, even if the choice is weighted—quite significantly—towards the games end of the market.

Low cost cassette recorders are, if only by implication, simple affairs. This is usually fine for a computer, which requires only a simple sound signal. So once you've managed to set up or match a recorder to the computer (by setting correct tone and volume levels), you're away, and need only occasional maintenance to keep things running sweetly.

Well that's the theory anyway. In practice, everyone has no end of trouble getting the cassette recorder and computer combination right and keeping it that way. (See also pages 22 to 25.)

RECORDING QUALITY

One solution is to buy a proper data cassette recorder. These differ from ordinary audio cassette recorders in that electronics are used to clean up the signal transmission so the off and on pulses are much clearer.

Dedicated recorders, used for instance by the Commodore computers, do just this and offer the added advantage that a single, simple connection is all that need be made. But even dedicated recorders are not entirely fault-free, suffering as they do from the sort of problems which plague all tape users.

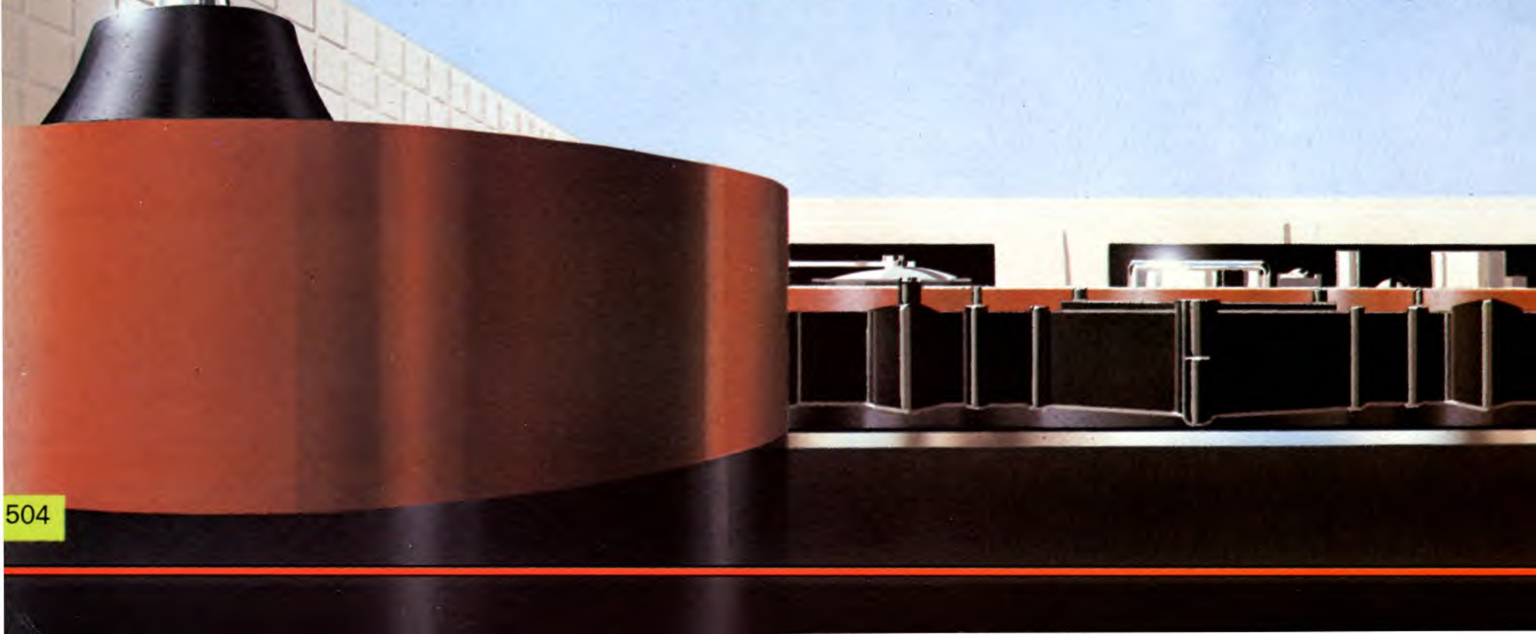
The one thing that is important when using tape is to maximize recording quality, as this is the one area where you can actually exercise some control. Even from a good recording, a signal will deteriorate as it passes through a poor quality cassette recorder. The same is true transferring data from memory to tape: a poor recorder can hardly be expected to make a good recording on tape.

Another important consideration is tape quality. Basically, you should avoid the temptation of using any old tape—especially previously used or lengthy audio cassettes.

There's some conflict here: because of the way data is stored on tape (see below) tape systems have fairly limited capacity. Although it is possible to use lengthy audio cassettes the reliability of these is extremely suspect. C90 and C120 tapes are much thinner than the 'safe' C30 and are consequently prone to severe stretching if subjected to frequent use—or continual to-ing and fro-ing under the comparatively 'vicious' fast forward/rewind mechanisms of cheap cassette recorders.

This stretching causes 'drop outs' and wavering signals, either of which spells the end as far as the demanding requirements of a computer are concerned. At best this can result in difficult, temperamental loading, at worst the loss of data if SAVEing is being attempted.

The practical upper limit is a C30 cassette.



- CONVENIENCE AND LOW COST OF TAPE SYSTEMS
- IMPORTANCE OF RELIABILITY
- DATA TRANSMISSION RATES
- TAPE LOOP DEVICES
- SLOWNESS OF SERIAL ACCESS
- HOW DISK DRIVES WORK
- THE DISK OPERATING SYSTEM
- INTERFACING
- SOFTWARE CHOICE

Each side can hold fifteen minutes of program transmission time. This is close to the memory limit of the slow loading Commodore machines and enough for several programs at the faster transmission rates possible on other home computers. If used with *great* care, C60 tapes can also prove suitable.

The longest tape required for a single program would be about of 20 minute's running time for the Commodores. All the other machines can get away with using C15 tape: only 7 minutes is required on the BBC, 6 minutes for the Spectrum, and as little as 3 minutes for the 32K Dragon.

It makes a lot of sense, however, to keep to the shortest possible tapes so that each cassette can be set aside for a single program on each side—which, for security, may even be a duplicate. That way there's really no problem finding the start of either!

Data quality cassettes made specially for computer use are usually available in these short lengths—typically up to C15. Many so-called data tapes, if cheap, could be anything but. These really should be tested thoroughly. If you can find a good brand, stick with it!

TRANSMISSION RATE

Any tape-based problems become increasingly severe the greater the data transmission speed used by a computer. Transmission speed—how quickly information passes from computer to storage or vice versa—is usually referred to by a *baud* rating. A baud loosely equates to the number of bits transferred per second but the figure is actually based on a much more complicated assessment.

The higher the baud rate used for program transmission, the better the equipment and recording medium have to be. A tape which may prove suitable for the snail-pace Commodores may not work on any of the other machines. Fast LOAD routines, written into lengthy games may also pose problems.

TAPE LOOPS

To get round some of the problems associated with conventional tape cassettes, several types of tape loop system are now available. Notable amongst these is the Microdrive for the Spectrum—but endless tape cartridge units can be used on most other machines if the necessary connections can be made.

These devices are essentially tape recorders except that an endless loop of tape passes the read/write head. The tape travels only in one direction, but very, very quickly so that the start point of a particular program can be reached quickly and without actually having to rewind tape. This gives a considerable time saving and the single direction of movement means much less wear and tear on the tape itself.

With faster access times and storage capacities of around 100K, tape loop systems seem capable of countering many of the shortfalls of standard cassette systems.

But the software available in this form is, to say the least, restricted. And in no way do the data capacity or access speeds match the necessary requirements of serious educational or business applications.

SERIAL ACCESS

In any tape system, the initial transfer of data from the computer to the storage device is literally a continuous dump from memory. Information is SAVEd (and later LOADed back) sequentially—in other words, in start to finish order. To get the program back into memory, you have to find the starting point and reload from there. This is called serial access.

For games programs this is not much of a handicap



—you simply rewind to the start each time and off you go. But this approach is next to useless when the computer has to access and manipulate very large amounts of data quickly. And that's just the sort of thing it needs to do in any 'serious' business application.

In cases like this there's only one solution: a disk drive.

But loop devices do at least divide the length of tape into 'blocks' which have markers. And because much greater rotating speeds are used, it is possible for the computer to locate these markers very quickly, thus simulating the random access files which give disk drives such power.

DISK DRIVES

Disk drives are among the fastest devices for both recording and accessing data. In the latter respect, times can often be measured in thousandths of a second. Program or data SAVEs and LOADs are also very quick and this is a very endearing quality to those who hate the tedium of tape SAVEs and LOADs.

Disks are also much more reliable than tapes providing elementary precautions are taken. And this is important if you're talking about huge amounts of data being manipulated. But disk drive use imposes a whole new set of 'housekeeping' procedures which demands a level of meticulousness that is simply not required for tape systems.

Abuse these procedures, and the consequences can be rather more severe than anything that could happen to data stored on tape.

This is because disks are capable of storing phenomenal amounts of data: up to ten megabytes in some cases—that's ten million bytes! Usually, though, for home computers the figure is somewhat less: maybe only 150K or 170K in some cases. But that's still a lot of valuable data!

Disks are also extremely vulnerable to things like physical abuse (bending, scratching, dirt, etc) and humidity, as well as magnetic fields of strengths easily generated by a loudspeaker magnet or TV tube.

Tape is a much better proposition for long-term storage. In fact, most big businesses would keep their master files on tape rather than on disk, but use the latter in day to day work for speed.

Regular backing up or copying of datafiles and work files is therefore essential, the frequency of this operation depending very much on what you're prepared to re-do if something dreadful actually does happen!



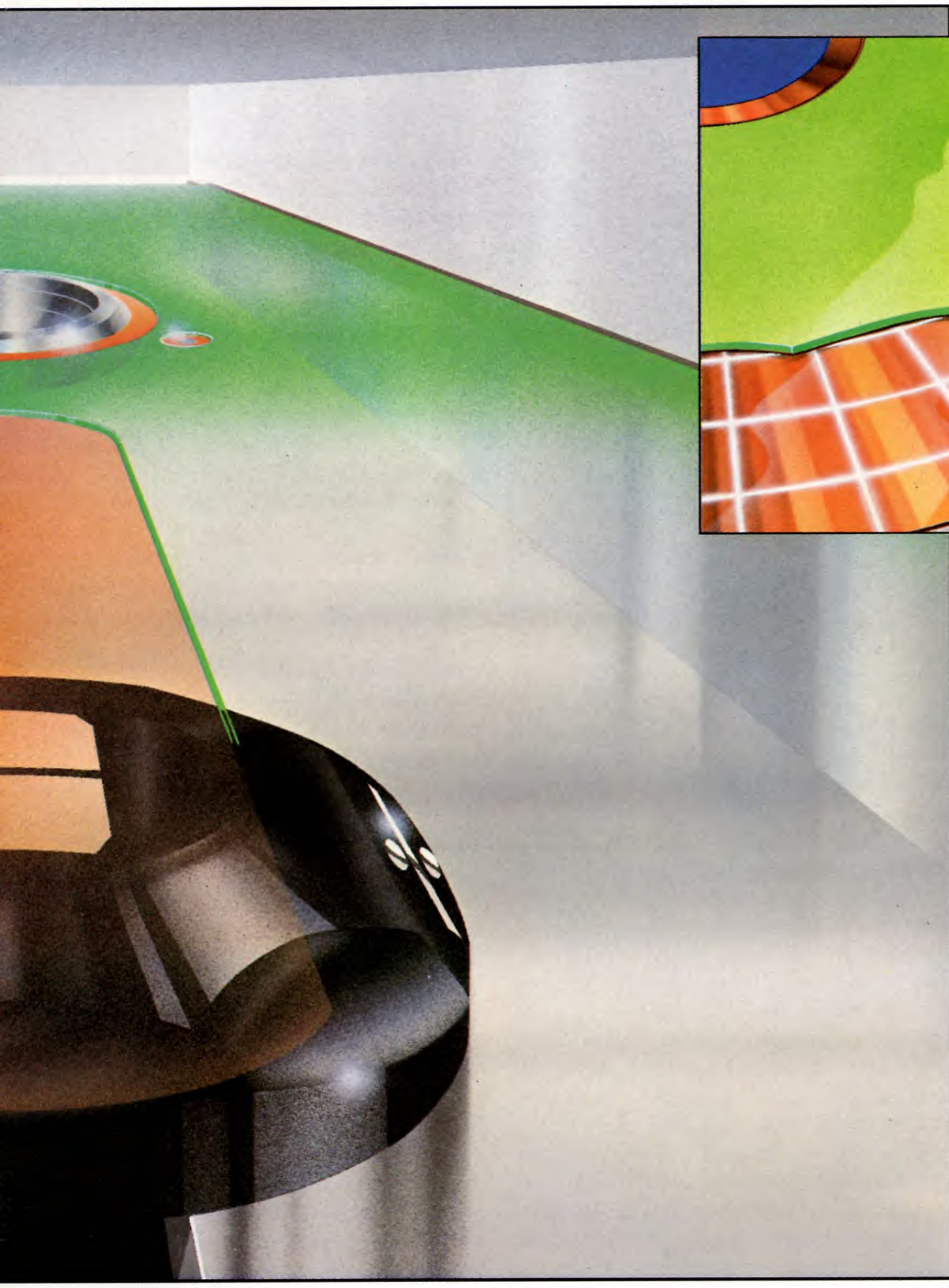
Q+A

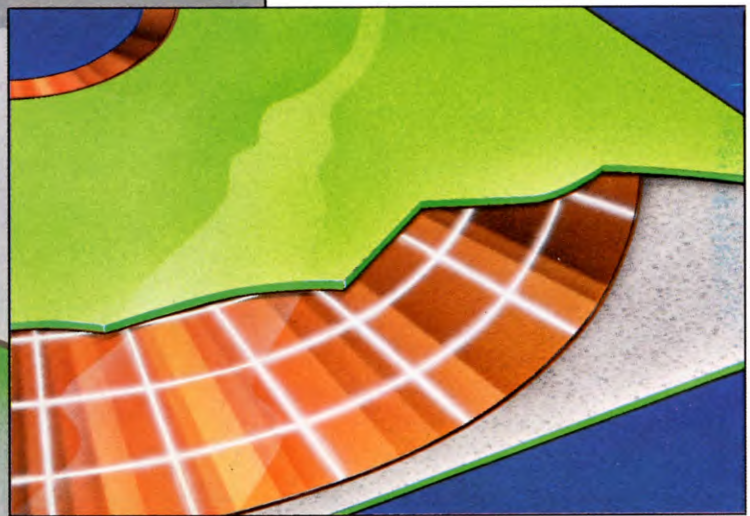
What advantages are there in using program cartridges rather than disk or tape based programs—and can I save my own programs on these?

The major benefit of using cartridge-based software is virtually instantaneous program loading. Cartridges are ROM (read only memory) devices which cannot be altered in any way—they are, in other words, a permanent means of program storage. You cannot record your own programs on these. Programs are called up under software control using a command which specifies the particular chip that is used.

On the BBC models, program-'chips' of this type may be inserted within the machine itself ready for use at any time.

Erasable programmable read-only memory, thankfully abbreviated to EPROM, can be used to create your own (semi) permanent chip storage devices. But setting up for this is expensive. Programs have to be 'burned' in electrically and erased with ultraviolet light.





A disk is formatted into track and sector 'blocks' prior to use. Each holds up to 256 bytes of information

HOW DISK DRIVES WORK

The working of a disk drive unit is not unlike a tape recorder. Instead of tape there is a circular piece of plastic, coated in much the same way but to very much higher standards. As in a tape recorder, this recording medium presses against a read/write head. But the difference here is that the head too is capable of movement, enabling it to track across and record on much of the surface of the disk as the latter rotates.

The surface of a disk is actually mapped out or *formatted* prior to use. Its area can be considered to consist of concentric rings or *tracks*, each of which is divided into a number of *sectors*. These sectors are in turn divided into *blocks* capable of holding up to 256 bytes of data each. These can be likened to the 'pigeon holes' of an array.

Information can be recorded sequentially, or at random, or in a pattern within these storage areas. Quite how it does this is controlled with a program via what is called a *disk operating system*, which is an essential part of the disk drive (see below).

The read/write head can be directed by the operating system to any sector on the disk surface, so offering filing capabilities quite unmatched by any tape system. It's rather like comparing the storage efficiency and capability of a single card folder with that of ten four-drawer office filing cabinets, with each drawer having its own indexed A–Z file—really!

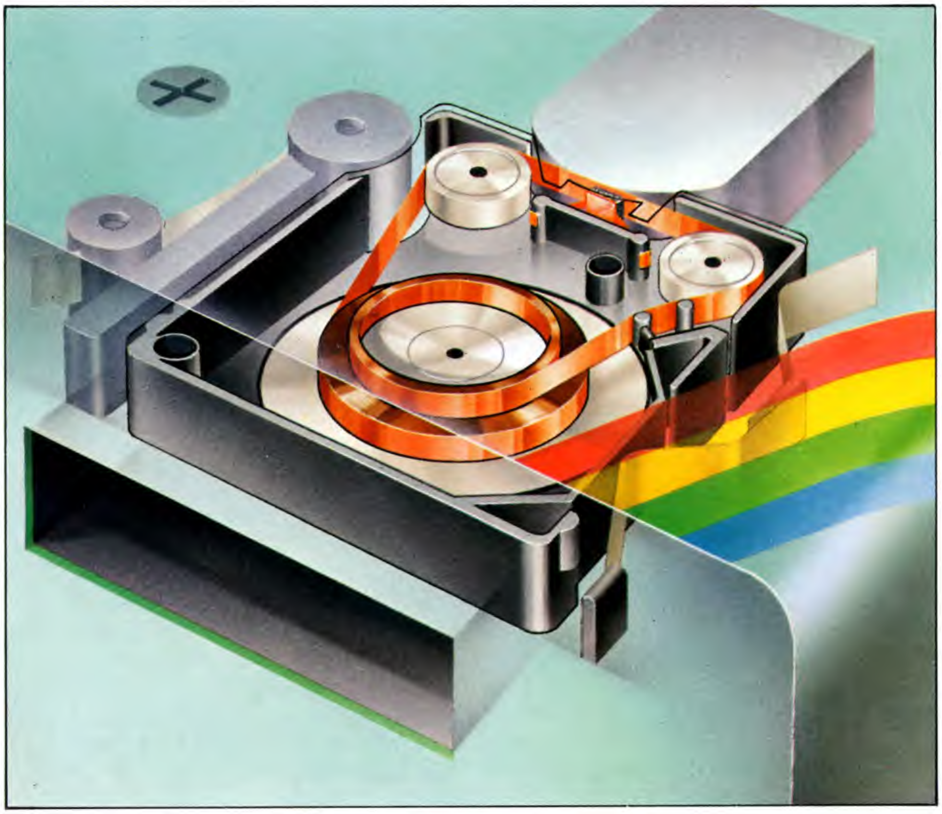
DISK TYPES

The disk itself, on flexible plastic—hence the description 'floppy disk'—is so vulnerable that it has to be protected by a self-lubricating semi-rigid envelope. The traditional size of disks for home computer use is 5.25 inches in diameter, compared to 8 inches for large-capacity business machines. However, newer drive units using 3 inch or $3\frac{1}{2}$ inch disks are becoming increasingly popular. These disks are encased in rigid plastic shields so they offer distinct handling advantages. And, with improvements in disk coating and recording technology, their capacity often far exceeds that of the traditional floppy.

The speed at which a disk rotates, its size, the type of disk coating, and the presence or not of another head on the reverse side of the disk all influence the rate and density of information storage. All therefore play an important part in the process of selecting a disk unit which is suitable for your computing needs.

Disks (and thus their drive units) fall into two basic categories: those designed to be read on a single side are, not surprisingly, called *single-sided*. Those which are read on both sides are referred to as *double-sided*.

Then there are three 'quality standards' to consider. Systems which can (or need) only read/write information slowly usually make



The Microdrive cassette, although small, holds about 100K of data in an endless tape loop

use of what are called *single density* disks. These typically have a fairly restricted storage capacity. The next grade up is *double density*, which permits more tightly packed information or information to be read and written more quickly. Then there's *quad density*, which allows the greatest rate and density of data storage—double the number of tracks are used, for instance. The actual method of recording pulses also differs on these.

Single density disks are usually batch-failed double density disks but you should never risk using these where double density is stipulated. For greater reliability, double-density disks may be used where only single-density are required.

Although, in a similar vein it is possible to use double-sided disks where single-sided ones are all that's required (though this is wasteful), you certainly can't work the other way round. And under no circumstances may a disk be flipped over in a single-sided disk unit so that use may be made of the reverse.

OPERATING SYSTEM

Most disk drives need special software to control them and, really, it is this part of a disk unit's make-up that decides how powerful and useful the system is. This software is essentially an operating system which manages all the necessary to-ing and fro-ing between computer memory and disk, which in most cases is described as 'dumb' in that it cannot act on its own. This type of operating system has to take residence somewhere—and that usually means losing precious RAM in the computer.

The dedicated disk unit for the Commodore 64 (and Vic) is slightly different in that it contains its own microprocessor, RAM and control circuits and is very much an 'intelligent' device which draws nothing more than information from the host computer.

You have to be careful with the other machines to use tried and tested disk operating systems and here the advice of a friendly but knowledgeable dealer—or fellow computer enthusiast—can be invaluable. But sometimes there simply isn't a choice to worry about!

INTERFACES AND CONNECTIONS

If you're thinking about upgrading to a disk unit, consider carefully whether or not the unit you choose offers full compatability with your machine. As well as a suitable disk operating system, you might find you need some method of *interfacing* the unit so it will work with your computer. This you'll have to check out carefully with your supplier. Again, knowledgeable advice is invaluable here, so see what fellow enthusiasts have done to get round any problems they've had in this respect.

DISK SOFTWARE

As far as disk-based software is concerned, only the Commodore 64 offers a really good choice of material. The disk drive really is slow by contemporary standards but is significantly faster than the ponderous tape unit that otherwise has to be used. The selection of disk-based games, educational and—particularly—business software really does transform the machine in a way tape-based software cannot match.

Quite a good selection of disk software is available for the BBC, though this is weighted towards educational and professional material.

Precious little is available for the Dragon and none for the Spectrum (this is understandable as its practical limits are neatly serviced by the Microdrive).

MAKING A CHOICE

Cost is obviously a significant factor when it comes to deciding between tape and disk systems. Much of the appeal of disk systems is lost on home users whose interest lies mainly in games software. Compared to cassette-based software, there's not much of choice of games or even general recreational software. What is available costs more than comparable cassette-based software. But there's every chance that the disk version is more sophisticated. Disk units are much more expensive than tape systems to buy—and typically costing as much as and often more than the computer itself. And then to this must be added, in most instances, a suitable disk operating system, interface, and leads.

Then there's the cost of the medium. Disks at the very least cost several times the price of the very best audio cassettes, four or five times the price of short-length 'data' tapes and about the same price as Microdrive cartridges. And the best quality 'lifetime guarantee' quad disks can cost four or five times the price of a single density one. But the point to note is that *all* of a disk's capacity can be used . . . and it is as quick to access the first program as it is the last.

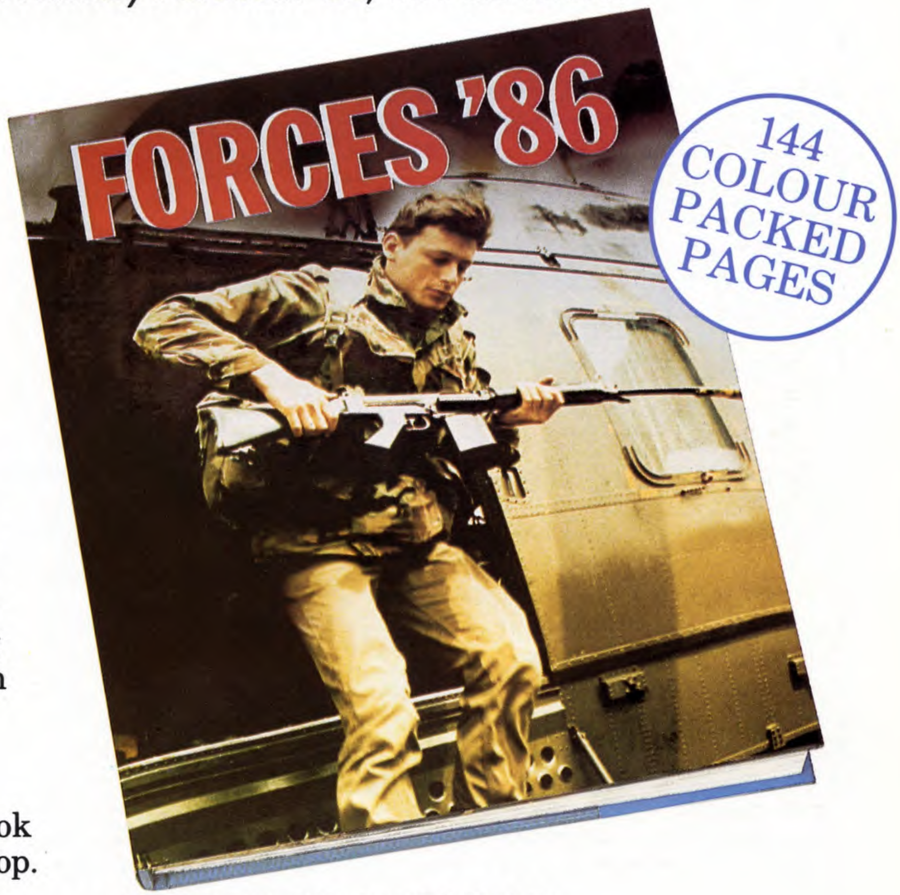
Beyond disk and tape units, it is feasible for home computers to access *hard disks* and to make use of *EPROM* devices (see Q&A box). Neither provides the flexibility of tape or disk but both have truly significant advantages. Hard disks, sometimes called *Winchesters*, offer incredible storage potential—2Ø megabytes is not uncommon—but they are very costly and quite unsuited to home uses.

REVEALED *the dramas and tensions, rumours and half truths – in war and peace.*

FIND OUT *what it means to have to fight, the price of keeping the peace. what faces British Military Forces in '86, and the future.*

FORCES '86

A fascinating and revealing insight into the who, what and why driving British Military Forces into 1986, and beyond.



FORCES '86

Looks at Britain's military scene as a whole. Personal accounts of marine training, combat flying and the submarine service compliment articles on war in space, a major European exercise and a historic and unique Regiment to present a stimulating and informative view of British armed forces today.

FORCES '86

Is an important and engrossing book that is not available in any book shop.

From the publishers of THE FALKLANDS WAR

SPECIAL OFFER ORDER FORM

FORCES '86

Simply fill in your name and address, cut out the coupon, put it in an envelope and post today.
No stamp is needed.

Address your envelope to:

Marshall Cavendish,
FREEPOST,
Lambourn Woodlands,
Hungerford, Berks. RG16 7BR

☐ **YES!** Please send me Forces '86, on approval, with an invoice for £7.95 (plus 95p postage and packing).

Name____________________________________

Address____________________________________

____

Tel. No.____________________________________

Signature____________________________(I am over 18)

GUARANTEE

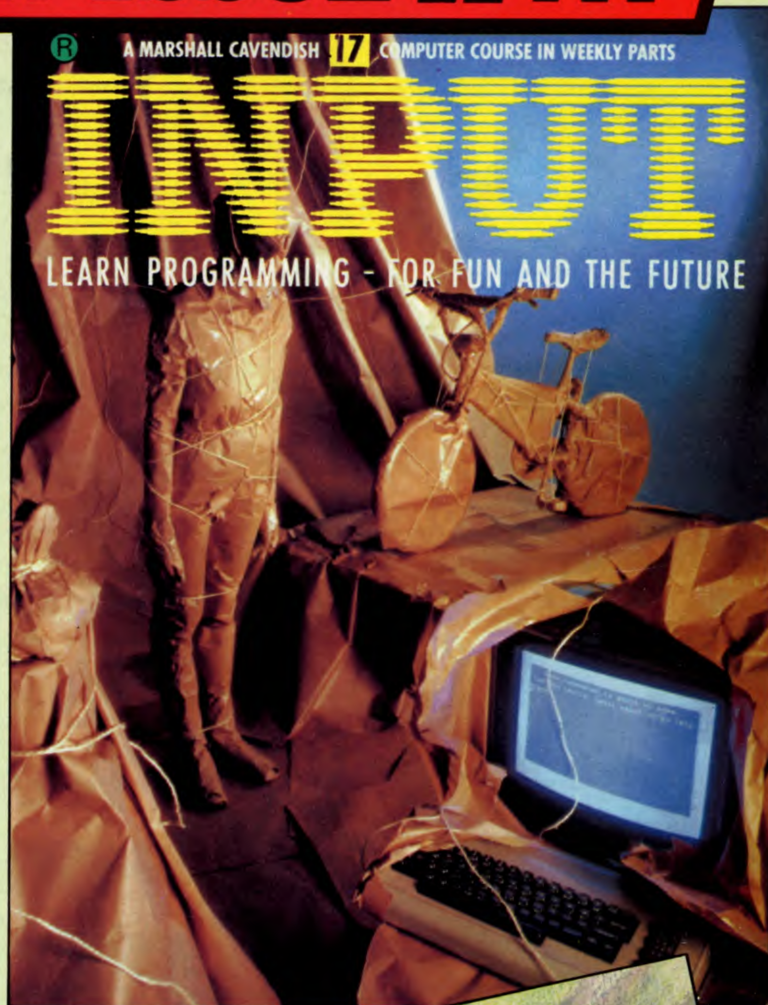
- Marshall Cavendish guarantees that if you are not entirely satisfied with your book and return it within 14 days, you will owe nothing.
- As a subscriber to Forces '86 you will be advised of, and entitled to receive on approval, any further annual publications.

F86/IP

Registered London 385817 BF12

ASSOCIATION OF MAIL ORDER PUBLISHERS

COMING IN ISSUE 17 . . .



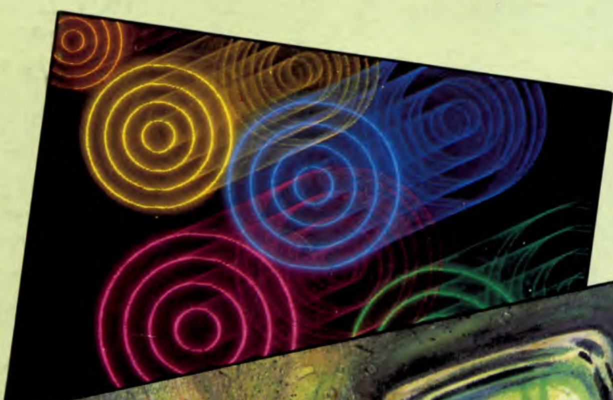
❑ ***Get started on computer card games. The first of three articles on setting up a complete game of PONTOON shows how to MAKE THE PACK AND SHUFFLE.***

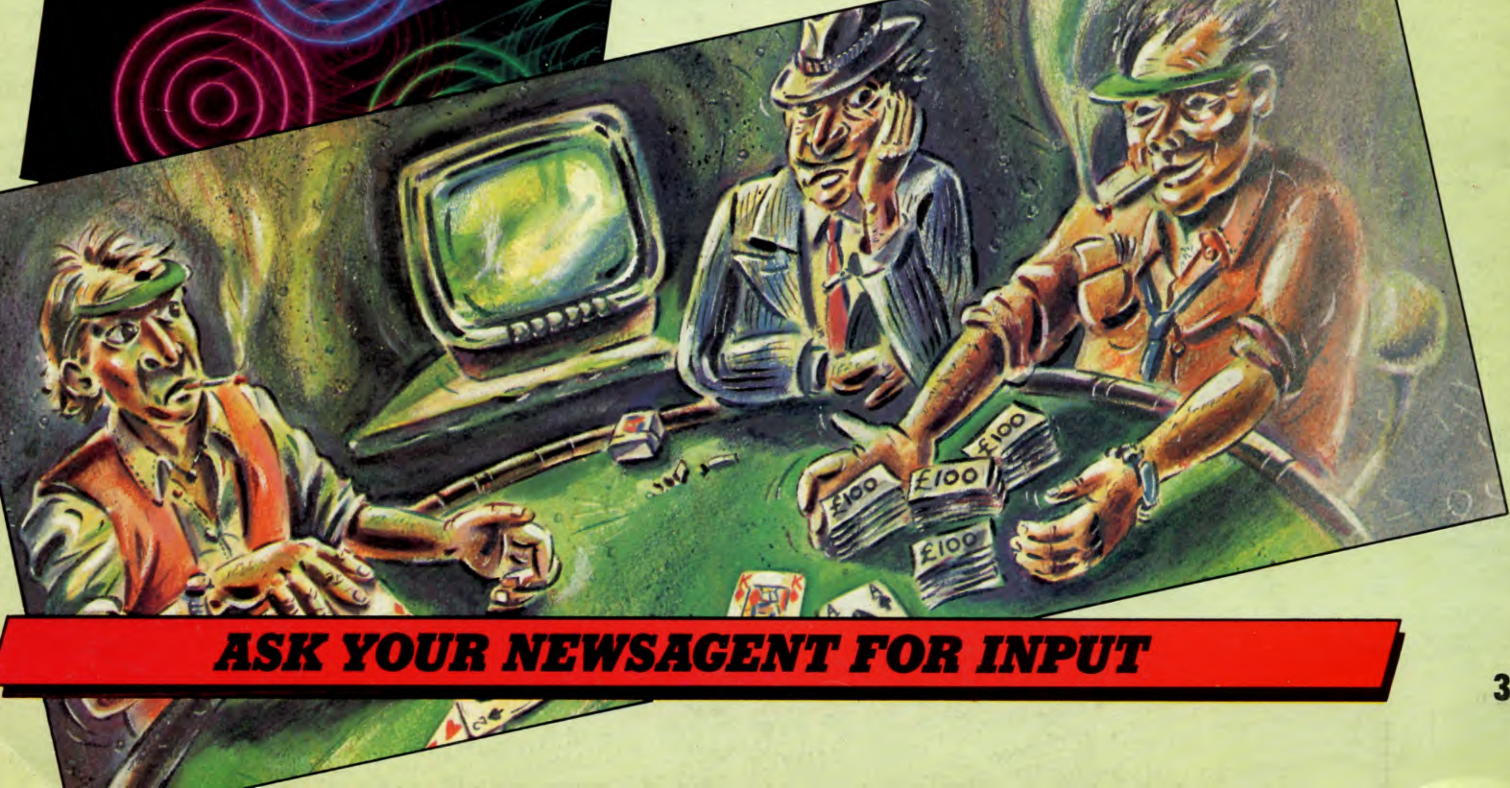
❑ ***Say 'computer graphics', and most people think of WIREFRAME DRAWINGS. Learn how you can generate them with BASIC drawing commands.***

❑ ***Complete your UDG picture by adding NEW CHARACTERS. You'll also discover how to save the image to memory, and how to animate the picture.***

❑ ***If you or someone you know is bemused by IMPERIAL/METRIC CONVERSIONS, there's a handy program that does them automatically.***

❑ ***For COMMODORE users, there is also a complete MACHINE CODE TRACE PROGRAM that you can use to check for bugs in other running programs.***





ASK YOUR NEWSAGENT FOR INPUT